

c o n f e r e n c e

proceedings

**USENIX 1996 Annual
Technical Conference**

*San Diego, California
January 22–26, 1996*



The UNIX® and Advanced
Computing Systems Professional
and Technical Association

For additional copies of these proceedings, write:

USENIX Association
2560 Ninth Street, Suite 215
Berkeley, CA 94710 USA

The price is \$32 for members and \$40 for nonmembers.
Outside the USA and Canada, please add
\$18 per copy for postage (via air printed matter).

Past USENIX Technical Conferences

1995 New Orleans	1988 Summer San Francisco
1994 Summer Boston	1988 Winter Dallas
1994 Winter San Francisco	1987 Summer Phoenix
1993 Summer Cincinnati	1987 Winter Washington, DC
1993 Winter San Diego	1986 Summer Atlanta
1992 Summer San Antonio	1986 Winter Denver
1992 Winter San Francisco	1985 Summer Portland
1991 Summer Nashville	1985 Winter Dallas
1991 Winter Dallas	1984 Summer Salt Lake City
1990 Summer Anaheim	1984 Winter Washington, DC
1990 Winter Washington, DC	1983 Summer Toronto
1989 Summer Baltimore	1983 Winter San Diego
1989 Winter San Diego	

1996 © Copyright by The USENIX Association
All Rights Reserved.

This volume is published as a collective work. Rights to individual papers remain with the author or the author's employer. Permission is granted for the noncommercial reproduction of the complete work for educational or research purposes. USENIX acknowledges all trademarks herein.

ISBN 1-880446-76-6

Printed in the United States of America on 50% recycled paper, 10-15% post consumer waste.



The USENIX Association

**Proceedings of the
USENIX 1996 Annual Technical Conference**

**January 22- 26, 1996
San Diego, California, USA**

TABLE OF CONTENTS

USENIX 1996 Annual Technical Conference
January 22-26, 1996
San Diego, California

Wednesday, January 24

Keynote Address: Nature and Nurture: The Interplay of UNIX and Networking

Van Jacobson, Lawrence Berkeley National Laboratory

FILE SYSTEMS

11am-12:30pm

Session Chair: John Ousterhout, Sun Microsystems Laboratories

Scalability in the XFS File System	1
<i>Adam Sweeney, Don Doucette, Wei Hu, Curtis Anderson, Mike Nishimoto, and Geoff Peck, Silicon Graphics</i>	
A Comparison of FFS Disk Allocation Policies	15
<i>Keith A. Smith and Margo Seltzer, Harvard University</i>	
AFRAID--A Frequently Redundant Array of Independent Disks	27
<i>Stefan Savage, University of Washington; John Wilkes, Hewlett-Packard Laboratories</i>	

OS EXTENSIONS

2pm-3:30pm

Session Chair: Darrell Long, University of California, Santa Cruz

A Comparison of OS Extension Technologies.....	41
<i>Christopher Small and Margo Seltzer, Harvard University</i>	
An Extensible Protocol Architecture for Application-Specific Networking	55
<i>Marc E. Fiuczynski and Brian N. Bershad, University of Washington</i>	
Linux Device Driver Emulation in Mach	65
<i>Shantanu Goel and Dan Duchamp, Columbia University</i>	

MULTIMEDIA

4pm-5pm

Session Chair: Brent Welch, Sun Microsystems Laboratories

Calliope: A Distributed, Scalable Multimedia Server	75
<i>Andrew Heybey, Mark Sullivan, and Paul England, Bellcore</i>	
Simple Continuous Media Storage Server on Real-Time Mach	87
<i>Hiroshi Tezuka and Tatsuo Nakajima, Japan Advanced Institute of Science and Technology</i>	

Thursday, January 25

NETWORK SESSION

9am-10:30am

Session Chair: John Kohl, Atria Software

Eliminating Receive Livelock in an Interrupt-driven Kernel.....	99
<i>Jeffrey C. Mogul, Digital Equipment Corporation Western Research Laboratory; K. K. Ramakrishnan, AT&T Bell Laboratories</i>	
Implementation of IPv6 in 4.4 BSD.....	113
<i>Randall J. Atkinson, Daniel L. McDonald, and Bao G. Phan, Naval Research Laboratory; Craig W. Metz, Kaman Sciences Corporation; Kenneth C. Chin, Naval Research Laboratory</i>	
Supporting Mobility in MosquitoNet.....	127
<i>Mary Baker, Xinhua Zhao, Stuart Cheshire, and Jonathan Stone, Stanford University</i>	

WEB

11am-12:30pm

Session Chair: Christopher Small, Harvard University

World Wide Web Cache Consistency	141
<i>James Gwertzman and Margo Seltzer, Harvard University</i>	
A Hierarchical Internet Object Cache	153
<i>Anawat Chankhunthod, Peter B. Danzig, and Chuck Neerdaels, University of Southern California; Michael F. Schwartz and Kurt J. Worrell, University of Colorado, Boulder</i>	
Tracking and Viewing Changes on the Web	165
<i>Fred Douglis and Thomas Ball, AT&T Bell Laboratories</i>	

DISTRIBUTED SYSTEMS

4pm-6pm Session Chair: David Black, Open Software Foundation Research Institute

Implementation of a Reliable Remote Memory Pager	177
<i>Evangelos P. Markatos and George Dramitinos, Institute of Computer Science, FORTH, Crete</i>	
Solaris MC: A Multi Computer OS.....	191
<i>Yousef A. Khalidi, Jose M. Bernabeu, Vlada Matena, Ken Shirriff, and Moti Thadani, Sun Microsystems Laboratories</i>	
A New Approach to Distributed Memory Management in the Mach Microkernel	205
<i>Stephan Zeisset, Stefan Tritscher, and Martin Mairandres, Intel GmbH</i>	
Fault Tolerance in a Distributed CHORUS/MiX System	219
<i>Sunil Kittur, Online Media; Douglas Steel, ICL High Performance Systems; Francois Armand and Jim Lipkis, Chorus Systems</i>	

Friday, January 26

PROTOCOLS

9am-10:30am

Session Chair: Jonathan Smith, University of Pennsylvania

FLIPC: A Low Latency Messaging System for Distributed Real Time Environments	229
<i>David L. Black, Randall D. Smith, Steven J. Sears, and Randall W. Dean, Open Software Foundation Research Institute</i>	
An Analysis of Process and Memory Models to Support High-Speed Networking in a UNIX Environment.....	239
<i>B. J. Murphy, University of Cambridge; S. Zeadally and C. J. Adams, University of Buckingham</i>	
Zero-Copy TCP in Solaris	253
<i>Hsiao-keng Jerry Chu, SunSoft, Inc.</i>	

PERFORMANCE

11am-12:30pm

Session Chair: Miche Baker-Harvey, Digital Equipment Corporation

A Performance Comparison of UNIX Operating Systems on the Pentium.....	265
<i>Kevin Lai and Mary Baker, Stanford University</i>	
lmbench: Portable Tools for Performance Analysis	279
<i>Larry McVoy, Silicon Graphics; Carl Staelin, Hewlett-Packard Laboratories</i>	
Process Labeled Kernel Profiling: A New Facility to Profile System Activities	295
<i>Shingo Nishioka, Atsuo Kawaguchi, and Hiroshi Motoda, Advanced Research Laboratory, Hitachi Ltd.</i>	

POT POURRI

2pm-4pm

Session Chair: Matt Blaze, AT&T Bell Laboratories

Cut-and-Paste File-Systems: Integrating Simulators and File-Systems	307
<i>Peter Bosch and Sape J. Mullender, Universiteit Twente, Netherlands</i>	
Predicting File-System Actions From Prior Events.....	319
<i>Thomas M. Kroeger and Darrell D. E. Long, University of California, Santa Cruz</i>	
Transparent Fault Tolerance for Parallel Applications on Networks of Workstations	329
<i>Daniel J. Scales, Digital Equipment Corporation Western Research Laboratory; Monica S. Lam, Stanford University</i>	
Why Use a Fishing Line When you Have a Net? An Adaptive Multicast Data Distribution Protocol.....	343
<i>Jeremy R. Cooperstock and Steve Kotsopoulos, University of Toronto</i>	

ACKNOWLEDGMENTS

Program Chair

Robert Gray, *US WEST Advanced Technologies*

Program Committee

Eric Allman, *Pangaea Reference Systems*
Miche Baker-Harvey, *Digital Equipment Corporation*
David Black, *Open Software Foundation Research Institute*
Matt Blaze, *AT&T Bell Laboratories*
John Kohl, *Atria Software*
Darrell Long, *University of California, Santa Cruz*
John Ousterhout, *Sun Microsystems Laboratories*
Christopher Small, *Harvard University*
Jonathan Smith, *University of Pennsylvania*
Carl Staelin, *Hewlett-Packard*
Brent Welch, *Sun Microsystems Laboratories*

External Reviewers

Michael Allen
Trevor Blackwell
Jay Bruce
Michael Cain
Pei Cao
Brad Chen
Peter Chen
Fred Douglis
Mike Durian
Yaz Endo
Dawson Engler
Brian Field
Steve Gaede
Jim Gettys
Richard Golding
Ted Haining
Jeffrey Harris
Mike Hibler
Linus Kamb
Berry Kercheval
Thomas Kroeger
William Lees
Sam Leffler
Jay Lepreau
Wei Liu
Gerald Robert Malan
Lynda McGinley
Marshall Kirk McKusick
Dylan McNamee
Larry McVoy
Jeffrey Mogul
Robert Morris
Balakumar Nagarajan
Mike O'Dell

Benjamin Reed

Eric Rehm
David R Richardson
David Rosenthal
Andy Rudoff
Roy D'Souza
Stefan Savage
Vernon Schryver
Margo Seltzer
Jonathan Shapiro
Tony Sloane
Keith Smith
Mark Swanson
Adam Sweeney
Jim Teague
Win Treese
Geoffrey M. Voelker
William Waite
Terri Watson
Alec Wolman

Invited Talks and Guru-Is-In Coordinators

Mary Baker, *Stanford University*
Ed Gould, *Digital Equipment Corporation*

Works in Progress Coordinator

Peg Schafer, *Harvard University*

USENIX Board Liaison

Eric Allman, *Pangaea Reference Systems*

Terminal Room

Gretchen Phillips, *University at Buffalo*

Tutorial Program

Daniel V. Klein, *USENIX Association*

Vendor Display

Zanna Knight, *USENIX Association*

Administration

The USENIX Association Staff

USENIX Marketing Director

Zanna Knight, *USENIX Association*

USENIX Executive Director

Ellie Young, *USENIX Association*

USENIX Meeting Planner

Judy DesHarnais, *USENIX Association*

PREFACE

Welcome to San Diego and the 1996 USENIX Technical Conference!

We are proud to present 28 outstanding papers on contemporary topics. A number of the papers will raise the standards for performance (networks, web caching, filesystems, tcp), other papers will help you measure and compare operating systems, and a few papers are about new, novel ways of getting your work done. Not included in these Proceedings are the Works-In-Progress (WIP) session, which is dedicated to research not yet complete and to practical, useful tools. In parallel, we have 10 Invited Talks and Panel Sessions to compete for your attention. We believe you will find the material interesting and useful. Enjoy!

I would like to thank all of the authors of the 130 submissions from 18 countries, 36 universities and over 30 companies. The time and effort describing your research is appreciated. It was difficult choosing the conference papers. However, a number of works will appear in other forums such as the WIP session, future USENIX conferences or other technical conferences. We as a committee strived to provide detailed feedback to improve every submission. As you attend this conference, and later read these proceedings, please consider your work as a future conference submission.

I would like to thank all of the people that helped make this conference possible of which I can only name a few; the rest will remain unsung heroes. The USENIX staff including Ellie Young, Judy DesHarnais, Zanna Knight and Carolyn Carr kept these conference productions running like clockwork. Evi Nemeth gets credit for the local arrangements for the program committee meeting and scribing. Thanks for the excellent, detailed reviews from the external readers (separately listed). The Invited Talk Coordinators Ed Gould and Mary Baker have complemented the refereed track with some fascinating speakers. I would like to thank my employer, US WEST, for supporting me during this project.

Finally, it was an honor and a pleasure working with my 11 Program Committee members. Thanks for the hard work this summer in Boulder and for working on the team.

Robert Gray, Program Chair
December 1995

AUTHOR INDEX

Adams, C. J.	239	Matena, Vlada	191
Anderson, Curtis	1	McDonald, Daniel L.	113
Armand, Francois	219	McVoy, Larry	279
Atkinson, Randall J.	113	Metz, Craig W.	113
Baker, Mary	127, 265	Mogul, Jeffrey C.	99
Ball, Thomas	165	Motoda, Hiroshi	295
Bernabeu, Jose M.	191	Mullender, Sape J.	307
Bershad, Brian N.	55	Murphy, B. J.	239
Black, David L.	229	Nakajima, Tatsuo	87
Bosch, Peter	307	Neerdaels, Chuck	153
Chankhunthod, Anawat	153	Nishimoto, Mike	1
Cheshire, Stuart	127	Nishioka, Shingo	295
Chin, Kenneth C.	113	Phan, Bao G.	113
Chu, Hsiao-Keng Jerry	253	Peck, Geoff	1
Cooperstock, Jeremy R.	343	Ramakrishnan, K. K.	99
Danzig, Peter B.	153	Savage, Stefan	27
Dean, Randall W.	229	Scales, Daniel J.	329
Doucette, Don	1	Schwartz, Michael F.	153
Douglis, Fred	165	Sears, Steven J.	229
Dramitinos, George	177	Seltzer, Margo	15, 41, 141
Duchamp, Dan	65	Shirriff, Ken	191
England, Paul	75	Small, Christopher	41
Fiuczynski, Marc E.	55	Smith, Keith A.	15
Goel, Shantanu	65	Smith, Randall D.	229
Gwertzman, James	141	Staelin, Carl	279
Heybey, Andrew	75	Steel, Douglas	219
Hu, Wei	1	Stone, Jonathan	127
Kawaguchi, Atsuo	295	Sullivan, Mark	75
Khalidi, Yousef A.	191	Sweeney, Adam	1
Kittur, Sunil	219	Tezuka, Hiroshi	87
Kotsopoulos, Steve	343	Thadani, Moti	191
Kroeger, Thomas M.	319	Tritscher, Stefan	205
Lai, Kevin	265	Wilkes, John	27
Lam, Monica S.	329	Worrell, Kurt J.	153
Lipkis, Jim	219	Zeadally, S.	239
Long, Darrell D. E.	319	Zeisset, Stephan	205
Mairandres, Martin	205	Zhao, Xinhua	127
Markatos, Evangelos P.	177		

Scalability in the XFS File System

Adam Sweeney, Doug Doucette, Wei Hu, Curtis Anderson, Mike Nishimoto, and Geoff Peck

Silicon Graphics, Inc.

Abstract

In this paper we describe the architecture and design of a new file system, XFS, for Silicon Graphics' IRIX operating system. It is a general purpose file system for use on both workstations and servers. The focus of the paper is on the mechanisms used by XFS to scale capacity and performance in supporting very large file systems. The large file system support includes mechanisms for managing large files, large numbers of files, large directories, and very high performance I/O.

In discussing the mechanisms used for scalability we include both descriptions of the XFS on-disk data structures and analyses of why they were chosen. We discuss in detail our use of B+ trees in place of many of the more traditional linear file system structures.

XFS has been shipping to customers since December of 1994 in a version of IRIX 5.3, and we are continuing to improve its performance and add features in upcoming releases. We include performance results from running on the latest version of XFS to demonstrate the viability of our design.

1. Introduction

XFS is the next generation local file system for Silicon Graphics' workstations and servers. It is a general purpose Unix file system that runs on workstations with 16 megabytes of memory and a single disk drive and also on large SMP network servers with gigabytes of memory and terabytes of disk capacity. In this paper we describe the XFS file system with a focus on the mechanisms it uses to manage large file systems on large computer systems.

The most notable mechanism used by XFS to increase the scalability of the file system is the pervasive use of B+ trees [Comer79]. B+ trees are used for tracking free extents in the file system rather than bitmaps. B+ trees are used to index directory entries rather than using linear lookup structures. B+ trees are used to manage file extent maps that overflow the number of direct pointers kept in the inodes. Finally, B+ trees are used to keep track of dynamically allocated inodes scattered throughout the file system. In addition, XFS

uses an asynchronous write ahead logging scheme for protecting complex metadata updates and allowing fast file system recovery after a crash. We also support very high throughput file I/O using large, parallel I/O requests and DMA to transfer data directly between user buffers and the underlying disk drives. These mechanisms allow us to recover even very large file systems after a crash in typically less than 15 seconds, to manage very large file systems efficiently, and to perform file I/O at hardware speeds that can exceed 300 MB/sec.

XFS has been shipping to customers since December of 1994 in a version of IRIX 5.3, and XFS will be the default file system installed on all SGI systems starting with the release of IRIX 6.2 in early 1996. The file system is stable and is being used on production servers throughout Silicon Graphics and at many of our customers' sites.

In the rest of this paper we describe why we chose to focus on scalability in the design of XFS and the mechanisms that are the result of that focus. We start with an explanation of why we chose to start from scratch rather than enhancing the old IRIX file system. We next describe the overall architecture of XFS, followed by the specific mechanisms of XFS which allow it to scale in both capacity and performance. Finally, we present performance results from running on real systems to demonstrate the success of the XFS design.

2. Why a New File System?

The file system literature began predicting the coming of the "I/O bottleneck" years ago [Ousterhout90], and we experienced it first hand at SGI. The problem was not the I/O performance of our hardware, but the limitations imposed by the old IRIX file system, EFS [SGI92]. EFS is similar to the Berkeley Fast File System [McKusick84] in structure, but it uses extents rather than individual blocks for file space allocation and I/O. EFS could not support file systems greater than 8 gigabytes in size, files greater than 2 gigabytes in size, or give applications access to the full I/O bandwidth of the hardware on which they were

running. EFS was not designed with large systems and file systems in mind, and it was faltering under the pressure coming from the needs of new applications and the capabilities of new hardware. While we considered enhancing EFS to meet these new demands, the required changes were so great that we decided it would be better to replace EFS with an entirely new file system designed with these new demands in mind.

One example of a new application that places new demands on the file system is the storage and retrieval of uncompressed video. This requires approximately 30 MB/sec of I/O throughput for a single stream, and just one hour of such video requires 108 gigabytes of disk storage. While most people work with compressed video to make the I/O throughput and capacity requirements easier to manage, many customers, for example those involved in professional video editing, come to SGI looking to work with uncompressed video. Another video storage example that influenced the design of XFS is a video on demand server, such as the one being deployed by Time Warner in Orlando. These servers store thousands of compressed movies. One thousand typical movies take up around 2.7 terabytes of disk space. Playing two hundred high quality 0.5 MB/sec MPEG streams concurrently uses 100 MB/sec of I/O bandwidth. Applications with similar requirements are appearing in database and scientific computing, where file system scalability and performance is sometimes more important than CPU performance. The requirements we derived from these applications were support for terabytes of disk space, huge files, and hundreds of megabytes per second of I/O bandwidth.

We also needed to ensure that the file system could provide access to the full capabilities and capacity of our hardware, and to do so with a minimal amount of overhead. This meant supporting systems with multiple terabytes of disk capacity. With today's 9 gigabyte disk drives it only takes 112 disk drives to surpass 1 terabyte of storage capacity. These requirements also meant providing access to the large amount of disk bandwidth that is available in such high capacity systems. Today, SGI's high end systems have demonstrated sustained disk bandwidths in excess of 500 megabytes per second. We needed to make that bandwidth available to applications using the file system. Finally, these requirements meant doing all of this without using unreasonable portions of the available CPU and memory on the systems.

3. Scalability Problems Addressed by XFS

In designing XFS, we focused in on the specific problems with EFS and other existing file systems that we felt we needed to address. In this section we consider several of the specific scalability problems addressed in the design of XFS and why the mechanisms used in other file systems are not sufficient.

Slow Crash Recovery

A file system with a crash recovery procedure that is dependent on the file system size cannot be practically used on large systems, because the data on the system is unavailable for an unacceptably long period after a crash. EFS and file systems based on the BSD Fast File System [McKusick84] falter in this area due to their dependence on a file system scavenger program to restore the file system to a consistent state after a crash. Running fsck over an 8 gigabyte file system with a few hundred thousand inodes today takes a few minutes. This is already too slow to satisfy modern availability requirements, and the time it takes to recover in this way only gets worse when applied to larger file systems with more files. Most recently designed file systems apply database recovery techniques to their metadata recovery procedure to avoid this pitfall.

Inability to Support Large File Systems

We needed a file system that could manage even petabytes of storage, but all of the file systems we know of are limited to either a few gigabytes or a few terabytes in size. EFS is limited to only 8 gigabytes in size. These limitations stem from the use of data structures that don't scale, for example the bitmap in EFS, and from the use of 32 bit block pointers throughout the on-disk structures of the file system. The 32 bit block pointers can address at most 4 billion blocks, so even with an 8 kilobyte block size the file system is limited to a theoretical maximum of 32 terabytes in size.

Inability to Support Large, Sparse Files

None of the file systems we looked at support full 64 bit, sparse files. EFS did not support sparse files at all. Most others use the block mapping scheme created for FFS. We decided early on that we would manage space in files with variable length extents (which we will describe later), and the FFS style scheme does not work with variable length extents. Entries in the FFS block map point to individual blocks in the file, and up to three levels of indirect blocks can be used to

track blocks throughout the file. This scheme requires that all entries in the map point to extents of the same size. This is because it does not store the offset of each entry in the map with the entry, and thus forces each entry to be in a fixed location in the tree so that it can be found. Also, a 64 bit file address space cannot be supported at all without adding more levels of indirection to the FFS block map.

Inability to Support Large, Contiguous Files

Another problem is that the mechanisms in many other file systems for allocating large, contiguous files do not scale well. Most, including EFS, use linear bitmap structures for tracking free and allocated blocks in the file system. Finding large regions of contiguous space in such bitmaps in large file systems is not efficient. For EFS this has become a significant bottleneck in the performance of writing newly allocated files. For other file systems, for example FFS, this has not been a problem up to this point, because they do not try very hard to allocate files contiguously. Not doing so, however, can have bad implications for the I/O performance of accessing files in those file systems [Seltzer95].

Inability to Support Large Directories

Another area which has not been addressed by other Unix file systems is support for directories with more than a few thousand entries. While some, for example Episode [Chutani92] and VxFS [Veritas95], at least speed up searching for entries within a directory block via hashing, most file systems use directory structures which require a linear scan of the directory blocks in searching for a particular file. The lookup and update performance of these unindexed formats degrades linearly with the size of the directory. Others use in-memory hashing schemes layered over simple on-disk structures [Hitz94]. These in memory schemes work well to a point, but in very large directories they require a large amount of memory. This problem has been addressed in some non-Unix file systems, like NTFS [Custer94] and Cedar [Hagmann87], by using B trees to index the entries in the directory.

Inability to Support Large Numbers of Files

While EFS and other file system can theoretically support very large numbers of files in a file system, in practice they do not. The reason is that the number of inodes allocated in these file systems is fixed at the time the file system is created. Choosing a very large number of inodes up front wastes the space allocated

to those inodes when they are not actually used. The real number of files that will reside in a file system is rarely known at the time the file system is created. Being forced to choose makes the management of large file systems more difficult than it should be. Episode [Chutani92] and VxFS [Veritas95] both solve this problem by allowing the number of inodes in the file system to be increased dynamically.

In summary, there are several problems with EFS and other file systems that we wanted to address in the design of XFS. While these problems may not have been important in the past, we believe the rules of file system design have changed. The rest of this paper describes XFS and the ways in which it solves the scalability problems described here.

4. XFS Architecture

Figure 1. gives a block diagram of the general structure of the XFS file system.

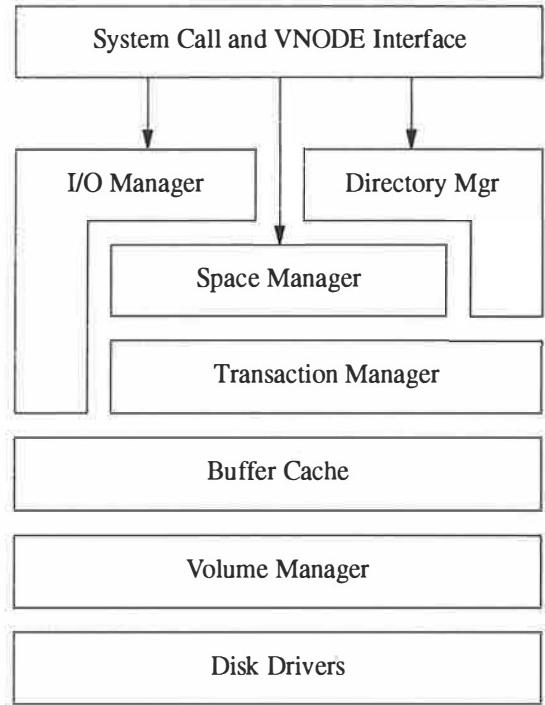


Figure 1. XFS Architecture

The high level structure of XFS is similar to a conventional file system with the addition of a transaction manager and a volume manager. XFS supports all of the standard Unix file interfaces and is entirely POSIX and XPG4 compliant. It sits below the vnode interface [Kleiman86] in the IRIX kernel and takes full

advantage of services provided by the kernel, including the buffer/page cache, the directory name lookup cache, and the dynamic vnode cache.

XFS is modularized into several parts, each of which is responsible for a separate piece of the file system's functionality. The central and most important piece of the file system is the space manager. This module manages the file system free space, the allocation of inodes, and the allocation of space within individual files. The I/O manager is responsible for satisfying file I/O requests and depends on the space manager for allocating and keeping track of space for files. The directory manager implements the XFS file system name space. The buffer cache is used by all of these pieces to cache the contents of frequently accessed blocks from the underlying volume in memory. It is an integrated page and file cache shared by all file systems in the kernel. The transaction manager is used by the other pieces of the file system to make all updates to the metadata of the file system atomic. This enables the quick recovery of the file system after a crash. While the XFS implementation is modular, it is also large and complex. The current implementation is over 50,000 lines of C code, while the EFS implementation is approximately 12,000 lines.

The volume manager used by XFS, known as XLV, provides a layer of abstraction between XFS and its underlying disk devices. XLV provides all of the disk striping, concatenation, and mirroring used by XFS. XFS itself knows nothing of the layout of the devices upon which it is stored. This separation of disk management from the file system simplifies the file system implementation, its application interfaces, and the management of the file system.

5. Storage Scalability

XFS goes to great lengths to efficiently support large files, large file systems, large numbers of files, and large directories. This section describes the mechanisms used to achieve such scalability in size.

5.1. Allocation Groups

XFS supports full 64 bit file systems. All of the global counters in the system are 64 bits in length. Block addresses and inode numbers are also 64 bits in length. To avoid requiring all structures in XFS to scale to the 64 bit size of the file system, the file system is partitioned into regions called allocation groups (AGs). These are somewhat similar to the cylinder groups in FFS, but AGs are used for scalability and parallelism rather than disk locality.

Allocation groups keep the size of the XFS data structures in a range where they can operate efficiently without breaking the file system into an unmanageable number of pieces. Allocation groups are typically 0.5 to 4 gigabytes in size. Each AG has its own separate data structures for managing the free space and inodes within its boundaries. Partitioning the file system into AGs limits the size of the individual structures used for tracking free space and inodes. The partitioning also allows the per-AG data structures to use AG relative block and inode pointers. Doing so reduces the size of those pointers from 64 to 32 bits. Like the limitations on the size of the region managed by the AG, this helps to keep the per-AG data structures to an optimal size.

Allocation groups are only occasionally used for disk locality. They are generally far too large to be of much use in this respect. Instead, we establish locality around individual files and directories. Like FFS, each time a new directory is created, we place it in a different AG from its parent. Once we've allocated a directory, we try to cluster the inodes in that directory and the blocks for those inodes around the directory itself. This works well for keeping directories of small files clustered together on disk. For large files, we try to allocate extents initially near the inode and afterwards near the existing block in the file which is closest to the offset in the file for which we are allocating space. That implies blocks will be allocated near the last block in the file for sequential writers and near blocks in the middle of the file for processes writing into holes. Files and directories are not limited to allocating space within a single allocation group, however. While the structures maintained within an AG use AG relative pointers, files and directories are file system global structures that can reference inodes and blocks anywhere in the file system.

The other purpose of allocation groups is to allow for parallelism in the management of free space and inode allocation. Previous file systems, like SGI's EFS, have single threaded block allocation and freeing mechanisms. On a large file system with large numbers of processes running, this can be a significant bottleneck. By making the structures in each AG independent of those in the other AGs, XFS enables free space and inode management operations to proceed in parallel throughout the file system. Thus, processes running concurrently can allocate space in the file system concurrently without interfering with each other.

5.2. Managing Free Space

Space management is key to good file system performance and scalability. Efficiently allocating and freeing space and keeping the file system from becoming fragmented are essential to good file system performance. XFS has replaced the block oriented bitmaps of other file systems with an extent oriented structure consisting of a pair of B+ trees for each allocation group. The entries in the B+ trees are descriptors of the free extents in the AG. Each descriptor consists of an AG relative starting block and a length. One of the B+ trees is indexed by the starting block of the free extents, and the other is indexed by the length of the free extents. This double indexing allows for very flexible and efficient searching for free extents based on the type of allocation being performed.

Searching an extent based tree is more efficient than a linear bitmap scan, especially for large, contiguous allocations. In searching a tree describing only the free extents, no time is wasted scanning bits for allocated blocks or determining the length of a given extent. According to our simulations, the extent based trees are just as efficient and more flexible than hierarchical bitmap schemes such as binary buddy bitmaps. Unfortunately, the results of those simulations have been lost, so we will have to settle here for an analytical explanation. Unlike binary buddy schemes, there are no restrictions on the alignment or size of the extents which can be allocated. This is why we consider the B+ trees more flexible. Finding an extent of a given size with the B+ tree indexed by free extent size, and finding an extent near a given block with the B+ tree indexed by extent starting block are both O(log N) operations. This is why we feel that the B+ trees are just as efficient as binary buddy schemes. The implementation of the allocation B+ trees is certainly more complex than normal or binary buddy bitmap schemes, but we believe that the combination of flexibility and performance we get from the B+ trees is worth the complexity.

5.3. Supporting Large Files

XFS provides a 64 bit, sparse address space for each file. The support for sparse files allows files to have holes in them for which no disk space is allocated. The support for 64 bit files means that there are potentially a very large number of blocks to be indexed for every file. In order to keep the number of entries in the file allocation map small, XFS uses an extent map rather than a block map for each file. Entries in the extent map are ranges of contiguous blocks allocated to the file. Each entry consists of the block offset of

the entry in the file, the length of the extent in blocks, and the starting block of the extent in the file system. In addition to saving space over a block map by compressing the allocation map entries for up to two million contiguous blocks into a single extent map entry, using extent descriptors makes the management of contiguous space within a file efficient.

Even with the space compression provided by an extent map, sparse files may still require large numbers of entries in the file allocation map. When the number of extents allocated to a file overflows the number that can fit immediately within an XFS inode, we use a B+ tree rooted within the inode to manage the extent descriptors. The B+ tree is indexed by the block offset field of the extent descriptors, and the data stored within the B+ tree are the extent descriptors. The B+ tree structure allows us to keep track of millions of extent descriptors, and, unlike an FFS style solution, it allows us to do so without forcing all extents to be of the same size. By storing the offset and length of each entry in the extent map in the entry, we gain the benefit of entries in the map which can point to variable length extents in exchange for a more complicated map implementation and less fan out at each level of the mapping tree (since our individual entries are larger we fit fewer of them in each indirect block).

5.4. Supporting Large Numbers of Files

In addition to supporting very large files, XFS supports very large numbers of files. The number of files in a file system is limited only by the amount of space in the file system to hold them. Rather than statically pre-allocating the inodes for all of these files, XFS dynamically allocates inodes as needed. This relieves the system administrator of having to guess the number of files that will be created in a given file system and of having to recreate the file system when that guess is wrong.

With dynamically allocated inodes, it is necessary to use some data structure for keeping track of where the inodes are located. In XFS, each allocation group manages the inodes allocated within its confines. Each AG uses a B+ tree to index the locations of the inodes within it. Inodes are allocated in chunks of sixty-four inodes. The inode allocation B+ tree in each AG keeps track of the locations of these chunks and whether each inode within a chunk is in use. The inodes themselves are not actually contained in the B+ tree. The B+ tree records only indicate where each chunk of inodes is located within the AG.

The inode allocation B+ trees, containing only the offset of each inode chunk along with a bit for each inode in the chunk, can each manage millions of inodes. This flexibility comes at the cost of additional complexity in the implementation of the file system. Deciding where to allocate new chunks of inodes and keeping track of them requires complexity that does not exist in other file systems. File system backup programs that need to traverse the inodes of the file system are also more complex, because they need to be able to traverse the inode B+ trees in order to find all of the inodes. Finally, having a sparse inode numbering space forced us to use 64 bit inode numbers, and this introduces a whole series of system interface issues for returning file identifiers to programs.

5.5. Supporting Large Directories

The millions of files in an XFS file system need to be represented in the file system name space. XFS implements the traditional Unix hierarchical name space. Unlike existing Unix file systems, XFS can efficiently support large numbers of files in a single directory. XFS uses an on-disk B+ tree structure for its directories.

The directory B+ tree is a bit different from the other B+ trees in XFS. The difference is that keys for the entries in the directory B+ tree, the names of the files in the directory, vary in length from 1 to 255 bytes. To hide this fact from the B+ tree index management code, the directory entry names are hashed to four byte values which are used as the keys in the B+ tree for the entries. The directory entries are kept in the leaves of the B+ tree. Each stores the full name of the entry along with the inode number for that entry. Since the hash function used for the directories is not perfect, the directory code must manage entries in the directory with duplicate keys. This is done by keeping entries with duplicate hash values next to each other in the tree. The use of fixed size keys in the interior nodes of the directory B+ tree simplifies the code for managing the B+ tree, but by making the keys non-unique via hashing we add significant complexity. We feel that the increased performance of the directory B+ trees that results from having fixed size keys, described below, is worth the increased complexity of the implementation.

The hashing of potentially large, variable length key values to small, constant size keys increases the breadth of the directory B+ trees. This reduces the height of the tree. The breadth of the B+ tree is increased by using small, constant sized keys in the interior nodes. This is because the interior nodes are

fixed in size to a single file system block, and compressing the keys allows more of them to fit into each interior node of the tree. This allows each interior node to have more children, thus increasing the breadth of the tree. In reducing the height of the tree, we reduce the number of levels that must be examined in searching for a given entry in the directory. The B+ tree structure makes lookup, create, and remove operations in directories with millions of entries practical. However, listing the contents of a directory with a million entries is still impractical due to the size of the resulting output.

5.6. Supporting Fast Crash Recovery

File systems of the size and complexity of XFS cannot be practically recovered by a process which examines the file system metadata to reconstruct the file system. In a large file system, examining the large amount of metadata will take too long. In a complex file system, piecing the on-disk data structures back together will take even longer. An example is the recovery of the XFS inode table. Since our inodes are not located in a fixed location, finding all of the inodes in the worst case where the inode B+ trees have been trashed can require scanning the entire disk for inodes. To avoid these problems, XFS uses a write ahead logging scheme that enables atomic updates of the file system. This scheme is very similar to the one described very thoroughly in [Hisgen93].

XFS logs all structural updates to the file system metadata. This includes inodes, directory blocks, free extent tree blocks, inode allocation tree blocks, file extent map blocks, AG header blocks, and the superblock. XFS does not log user data. For example, creating a file requires logging the directory block containing the new entry, the newly allocated inode, the inode allocation tree block describing the allocated inode, the allocation group header block containing the count of free inodes, and the superblock to record the change in its count of free inodes. The entry in the log for each of these items consists of header information describing which block or inode this is and a copy of the new image of the item as it should exist on disk.

Logging new copies of the modified items makes recovering the XFS log independent of both the size and complexity of the file system. Recovering the data structures from the log requires nothing but replaying the block and inode images in the log out to their real locations in the file system. The log recovery does not know that it is recovering a B+ tree. It only knows that it is restoring the latest images of some file

system blocks.

Unfortunately, using a transaction log does not entirely obsolete the use of file system scavenger programs. Hardware and software errors which corrupt random blocks in the file system are not generally recoverable with the transaction log, yet these errors can make the contents of the file system inaccessible. We did not provide such a repair program in the initial release of XFS, naively thinking that it would not be necessary, but our customers have convinced us that we were wrong. Without one, the only way to bring a corrupted file system back on line is to re-create it with mkfs and restore it from backups. We will be providing a scavenger program for all versions of XFS in the near future.

6. Performance Scalability

In addition to managing large amounts of disk space, XFS is designed for high performance file and file system access. XFS is designed to run well over large, striped disk arrays where the aggregate bandwidth of the underlying drives ranges in the tens to hundreds of megabytes per second.

The keys to performance in these arrays are I/O request size and I/O request parallelism. Modern disk drives have much higher bandwidth when requests are made in large chunks. With a striped disk array, this need for large requests is increased as individual requests are broken up into smaller requests to the individual drives. Since there are practical limits to individual request sizes, it is important to issue many requests in parallel in order to keep all of the drives in a striped array busy. The aggregate bandwidth of a disk array can only be achieved if all of the drives in the array are constantly busy.

In this section, we describe how XFS makes that full aggregate bandwidth available to applications. We start with how XFS works to allocate large contiguous files. Next we describe how XFS performs I/O to those files. Finally, we describe how XFS manages its metadata for high performance.

6.1. Allocating Files Contiguously

The first step in allowing large I/O requests to a file is to allocate the file as contiguously as possible. This is because the size of a request to the underlying drives is limited by the range of contiguous blocks in the file being read or written. The space manager in XFS goes to great lengths to ensure that files are allocated contiguously.

Delaying Allocation

One of the key features of XFS in allocating files contiguously is delayed file extent allocation. Delayed allocation applies lazy evaluation techniques to file allocation. Rather than allocating specific blocks to a file as it is written in the buffer cache, XFS simply reserves blocks in the file system for the data buffered in memory. A virtual extent is built up in memory for the reserved blocks. Only when the buffered data is flushed to disk are real blocks allocated for the virtual extent. Delaying the decision of which and how many blocks to allocate to a file as it is written provides the allocator with much better knowledge of the eventual size of the file when it makes its decision. When the entire file can be buffered in memory, the entire file can usually be to be allocated in a single extent if the contiguous space to hold it is available. For files that cannot be entirely buffered in memory, delayed allocation allows the files to be allocated in much larger extents than would otherwise be possible.

Delayed allocation fits well in modern file system design in that its effectiveness increases with the size of the memory of the system. As more data is buffered in memory, the allocator is provided with better and better information for making its decisions. Also, with delayed allocation, short lived files which can be buffered in memory are often never allocated any real disk blocks. The files are removed and purged from the file cache before they are pushed to disk. Such short lived files appear to be relatively common in Unix systems [Ousterhout85, Baker91], and delayed allocation reduces both the number of metadata updates caused by such files and the impact of such files on file system fragmentation.

Another benefit of delayed allocation is that files which are written randomly but have no holes can often be allocated contiguously. If all of the dirty data can be buffered in memory, the space for the randomly written data can be allocated contiguously when the dirty data is flushed out to disk. This is especially important for applications writing data with mapped files where random access is the norm rather than the exception.

Supporting Large Extents

To make the management of large amounts of contiguous space in a file efficient, XFS uses very large extent descriptors in the file extent map. Each descriptor can describe up to two million file system blocks, because we use 21 bits in the extent descriptor to store the length of the extent. Describing large numbers of blocks with a single extent descriptor eliminates the

CPU overhead of scanning entries in the extent map to determine whether blocks in the file are contiguous. We can simply read the length of the extent rather than looking at each entry to see if it is contiguous with the previous entry.

The extent descriptors used by XFS are 16 bytes in length. This is actually their compressed size, as the in memory extent descriptor needs 20 bytes (8 for file offset, 8 for the block number, and 4 for the extent length). Having such large extent descriptors forces us to have a smaller number of direct extent pointers in the inode than we would with smaller extent descriptors like those used by EFS (8 bytes total). We feel that this is a reasonable trade-off for XFS because of our focus on contiguous file allocation and the good performance of the indirect extent maps even when we do overflow the direct extents.

Supporting Variable Block Sizes

In addition to the above features for keeping disk space contiguous, XFS allows the file system block size to range from 512 bytes to 64 kilobytes on a per file system basis. The file system block size is the minimum unit of allocation and I/O request size. Thus, setting the block size sets the minimum unit of fragmentation in the file system. Of course, this must be balanced against the large amount of internal fragmentation that is caused by using very large block sizes. File systems with large numbers of small files, for example news servers, typically use smaller block sizes in order to avoid wasting space via internal fragmentation. File systems with large files tend to make the opposite choice and use large block sizes in order to reduce external fragmentation of the file system and their files' extents.

Avoiding File System Fragmentation

The work by Seltzer and Smith [Seltzer95] shows that long term file system fragmentation can degrade the performance of FFS file systems by between 5% and 15%. This fragmentation is the result of creating and removing many files over time. Even if all of the files are allocated contiguously, eventually, the remaining files are scattered about the disk. This fragments the file system's free space. Given the propensity of XFS for doing large I/O to contiguously allocated files, we could expect the degradation of XFS from its optimum performance to be even worse.

While XFS cannot completely avoid this problem, there are a few reasons why its impact is not as severe as it could be with XFS file systems. The first is the combination of delayed allocation and the allocation

B+ trees. In using the two together XFS makes requests for larger allocations to the allocator and is able to efficiently determine one of the best fitting extents in the file system for that allocation. This helps in delaying the onset of the fragmentation problem and reducing its performance impact once it occurs. A second reason is that XFS file systems are typically larger than EFS and FFS file systems. In a large file system, there is typically a larger amount of free space for the allocator to work with. In such a file system it takes much longer for the file system to become fragmented. Another reason is that file systems tend to be used to store either a small number of large files or a large number of small files. In a file system with a smaller number of large files, fragmentation will not be a problem, because allocating and deleting large files still leaves large regions of contiguous free space in the file system. In a file system containing mostly small files, fragmentation is not a big problem, because small files have no need for large regions of contiguous space. However, in the long term we still expect fragmentation to degrade the performance of XFS file systems, so we intend to add an on-line file system defragmentation utility to optimize the file system in the future.

6.2. Performing File I/O

Given a contiguously allocated file, it is the job of the XFS I/O manager to read and write the file in large enough requests to drive the underlying disk drives at full speed. XFS uses a combination of clustering, read ahead, write behind, and request parallelism in order to exploit its underlying disk array. For high performance I/O, XFS allows applications to use direct I/O to move data directly between application memory and the disk array using DMA. Each of these is described in detail below.

Handling Read Requests

To obtain good sequential read performance, XFS uses large read buffers and multiple read ahead buffers. By large read buffers, we mean that for sequential reads we use a large minimum I/O buffer size (typically 64 kilobytes). Of course, for files smaller than the minimum buffer size, we reduce the size of the buffers to match the files. Using a large minimum I/O size ensures that even when applications issue reads in small units the file system feeds the disk array requests that are large enough for good disk I/O performance. For larger application reads, XFS increases the read buffer size to match the application's request. This is very similar to the read

clustering scheme in SunOS [McVoy90], but it is more aggressive in using memory to improve I/O performance.

While large read buffers satisfy the need for large request sizes, XFS uses multiple read ahead buffers to increase the parallelism in accessing the underlying disk array. Traditional Unix systems have used only a single read ahead buffer at a time [McVoy90]. For sequential reads, XFS keeps outstanding two to three requests of the same size as the primary I/O buffer. The number varies because we try to keep three read ahead requests outstanding, but we wait until the process catches up a bit with the read ahead before issuing more. The multiple read ahead requests keep the drives in the array busy while the application processes the data being read. The larger number of read ahead buffers allows us to keep a larger number of underlying drives busy at once. Not issuing read ahead blindly, but instead waiting until the application catches up a bit, helps to keep sequential readers from flooding the drive with read ahead requests when the application is not keeping up with the I/O anyway.

Handling Write Requests

To get good write performance, XFS uses aggressive write clustering [McVoy90]. Dirty file data is buffered in memory in chunks of 64 kilobytes, and when a chunk is chosen to be flushed from memory it is clustered with other contiguous chunks to form a larger I/O request. These I/O clusters are written to disk asynchronously, so as data is written into the file cache many such clusters will be sent to the underlying disk array concurrently. This keeps the underlying disk array busy with a stream of large write requests.

The write behind used by XFS is tightly integrated with the delayed allocation mechanism described earlier. The more dirty data we can buffer in memory for a newly written file, the better the allocation for that file will be. This is balanced with the need to keep memory from being flooded with dirty pages and the need to keep I/O requests streaming out to the underlying disk array. This is mostly an issue for the file cache, however, so it will not be discussed in this paper.

Using Direct I/O

With very large disk arrays, it is often the case that the underlying I/O hardware can move data faster than the system's CPUs can copy that data into or out of the buffer cache. On these systems, the CPU is the bottleneck in moving data between a file and an application. For these situations, XFS provides what we call direct

I/O. Direct I/O allows a program to read or write a file without first passing the data through the system buffer cache. The data is moved directly between the user program's buffer and the disk array using DMA. This avoids the overhead of copying the data into or out of the buffer cache, and it also allows the program to better control the size of the requests made to the underlying devices. In the initial implementation of XFS, direct I/O was not kept coherent with buffered I/O, but this has been fixed in the latest version. Direct I/O is very similar to traditional Unix raw disk access, but it differs in that the disk addressing is indirectioned through the file extent map.

Direct I/O provides applications with access to the full bandwidth of the underlying disk array without the complexity of managing raw disk devices. Applications processing files much larger than the system's memory can avoid using the buffer cache since they get no benefit from it. Applications like databases that consider the Unix buffer cache a nuisance can avoid it entirely while still reaping the benefits of working with normal files. Applications with real-time I/O requirements can use direct I/O to gain fine grained control over the I/O they do to files.

The downsides of direct I/O are that it is more restrictive than traditional Unix file I/O and that it requires more sophistication from the application using it. It is more restrictive in that it requires the application to align its requests on block boundaries and to keep the requests a multiple of the block size in length. This often requires more complicated buffering techniques in the application that are normally handled by the Unix file cache. Direct I/O also requires more of the application in that it places the burden of making efficient I/O requests on the application. If the application writes a file using direct I/O and makes individual 4 kilobyte requests, the application will run much slower than if it made those same requests into the file cache where they could be clustered into larger requests. While direct I/O will never entirely replace traditional Unix file I/O, it is a useful alternative for sophisticated applications that need high performance file I/O.

Using Multiple Processes

Another barrier to high performance file I/O in many Unix file systems is the single threading inode lock used for each file. This lock ensures that only one process at a time may have I/O outstanding for a single file. This lock thwarts applications trying to increase the rate at which they can read or write a file using multiple processes to access the file at once.

XFS uses a more flexible locking scheme that allows multiple processes to read and write a file at once. When using normal, buffered I/O, multiple readers can access the file concurrently, but only a single writer is allowed access to the file at a time. The single writer restriction is due to implementation rather than architectural restrictions and will eventually be removed. When using direct I/O, multiple readers and writers can all access the file simultaneously. Currently, when using direct I/O and multiple writers, we place the burden of serializing writes to the same region of the file on the application. This differs from traditional Unix file I/O where file writes are atomic with respect to other file accesses, and it is one of the main reasons why we do not yet support multiple writers using traditional Unix file I/O.

Allowing parallel access to a file can make a significant difference in the performance of access to the file. When the bottleneck in accessing the file is the speed at which the CPU can move data between the application buffer and the buffer cache, parallel access to the file allows multiple CPUs to be applied to the data movement. When using direct I/O to drive a large disk array, parallel access to the file allows requests to be pipelined to the disk array using multiple processes to issue multiple requests. This feature is especially important for systems like IRIX that implement asynchronous I/O using threads. Without parallel file access, the asynchronous requests would be serialized by the inode lock and would therefore provide almost no performance benefit.

6.3. Accessing and Updating Metadata

The other side of file system performance is that of manipulating the file system metadata. For many applications, the speed at which files and directories can be created, destroyed, and traversed is just as important as file I/O rates. XFS attacks the problem of metadata performance on three fronts. The first is to use a transaction log to make metadata updates fast. The second is to use advanced data structures to change searches and updates from linear to logarithmic in complexity. The third is to allow parallelism in the search and update of different parts of the file system. We have already discussed the XFS data structures in detail, so this section will focus on the XFS transaction log and file system parallelism.

Logging Transactions

A problem that has plagued traditional Unix file systems is their use of ordered, synchronous updates to on-disk data structures in order to make those updates

recoverable by a scavenger program like fsck. The synchronous writes slow the performance of the metadata updates down to the performance of disk writes rather than the speed of today's fast CPUs [Rosenblum92].

XFS uses a write ahead transaction log to gather all the writes of an update into a single disk I/O, and it writes the transaction log asynchronously in order to decouple the metadata update rate from the speed of the disks. Other schemes such as log structured file systems [Rosenblum92], shadow paging [Hitz94], and soft updates [Ganger94] have been proposed to solve this problem, but we feel that write ahead logging provides the best trade-off among flexibility, performance, and reliability. This is because it provides us with the fast metadata updates and crash recovery we need without sacrificing our ability to efficiently support synchronous writing workloads, for example that of an NFS server [Sandberg85], and without sacrificing our desire for large, contiguous file support. However, an in depth analysis of write ahead logging or the tradeoffs among these schemes is beyond the scope of this paper.

Logging Transactions Asynchronously

Traditional write ahead logging schemes write the log synchronously to disk before declaring a transaction committed and unlocking its resources. While this provides concrete guarantees about the permanence of an update, it restricts the update rate of the file system to the rate at which it can write the log. While XFS provides a mode for making file system updates synchronous for use when the file system is exported via NFS, the normal mode of operation for XFS is to use an asynchronously written log. We still ensure that the write ahead logging protocol is followed in that modified data cannot be flushed to disk until after the data is committed to the on-disk log. Rather than keeping the modified resources locked until the transaction is committed to disk, however, we instead unlock the resources and pin them in memory until the transaction commit is written to the on-disk log. The resources can be unlocked once the transaction is committed to the in-memory log buffers, because the log itself preserves the order of the updates to the file system.

XFS gains two things by writing the log asynchronously. First, multiple updates can be batched into a single log write. This increases the efficiency of the log writes with respect to the underlying disk array [Hagmann87, Rosenblum92]. Second, the performance of metadata updates is normally made independent of the speed of the underlying drives. This

independence is limited by the amount of buffering dedicated to the log, but it is far better than the synchronous updates of older file systems.

Using a Separate Log Device

Under very intense metadata update workloads, the performance of the updates can still become limited by the speed at which the log buffers can be written to disk. This occurs when updates are being written into the buffers faster than the buffers can be written into the log. For these cases, XFS allows the log to be placed on a separate device from the rest of the file system. It can be stored on a dedicated disk or non-volatile memory device. Using non-volatile memory devices for the transaction log has proven very effective in high end OLTP systems [Dimino94]. It can be especially useful with XFS on an NFS server, where updates must be synchronous, in both increasing the throughput and decreasing the latency of metadata update operations.

Exploiting Parallelism

XFS is designed to run well on large scale shared memory multiprocessors. In order to support the parallelism of such a machine, XFS has only one centralized resource: the transaction log. All other resources in the file system are made independent either across allocation groups or across individual inodes. This allows inodes and blocks to be allocated and freed in parallel throughout the file system.

The transaction log is the most contentious resource in XFS. All updates to the XFS metadata pass through the log. However, the job of the log manager is very simple. It provides buffer space into which transactions can copy their updates, it writes those updates out to disk, and it notifies the transactions when the log writes complete. The copying of data into the log is easily parallelized by making the processor performing the transaction do the copy. As long as the log can be written fast enough to keep up with the transaction load, the fact that it is centralized is not a problem. However, under workloads which modify large amount of metadata without pausing to do anything else, like a program constantly linking and unlinking a file in a directory, the metadata update rate will be limited to the speed at which we can write the log to disk.

7. Experience and Performance Results

In this section we present results demonstrating the scalability and performance of the XFS file system. These results are not meant as a rigorous investigation

of the performance of XFS, but only as a demonstration of XFS's capabilities. We are continuing to measure and improve the performance of XFS as development of the file system proceeds.

7.1. I/O Throughput Test Results

Figures 2 and 3 contain the results of some I/O throughput tests run on a raw volume, XFS, and EFS. The results come from a test which measures the rate at which we can write a previously empty file (create), read it back (read), and overwrite the existing file (write). The number of drives over which the underlying volume is striped ranges from 3 to 57 in the test. The test system is an 8 CPU Challenge with 512 megabytes of memory. The test is run with three disks per SCSI channel, and each disk is capable of reading data sequentially at approximately 7 MB/sec and writing data sequentially at approximately 5.5 MB/sec. All tests are run on newly created file systems in order to measure the optimal performance of the file systems. All tests using EFS and XFS are using direct I/O and large I/O requests, and the tests using multiple threads are using the IRIX asynchronous I/O library with the given number of threads. Measurements for multiple, asynchronous threads with EFS are not given, because the performance of EFS with multiple threads is the same or worse as with one thread due to its single threaded (per file) I/O path. The test files are approximately 30 megabytes per disk in the volume in size, and for the raw volume tests we write the same amount of data to the volume itself. The stripe unit for the volumes is 84 kilobytes for the single threaded cases and 256 kilobytes for the multi-threaded cases. We have found these stripe units to provide the best performance for each of the cases in our experimentation.

We can draw several conclusions from this data. One is that XFS is capable of reading a file at nearly the full speed of the underlying volume. We manage to stay within 5-10% of the raw volume performance in all disk configurations when using an equivalent number of asynchronous I/O threads. Another interesting result is the parity of the create and write results for XFS versus the large disparity of the results for EFS. We believe that this demonstrates the efficiency of the XFS space allocator. Finally, the benefits of parallel file access are clearly demonstrated in these results. At the high end this makes a 55 MB/sec difference in the XFS read results. For writing and creating files it makes a 125 MB/sec difference. This is entirely because the parallel cases are capable of pipelining the drives with requests to keep them constantly busy

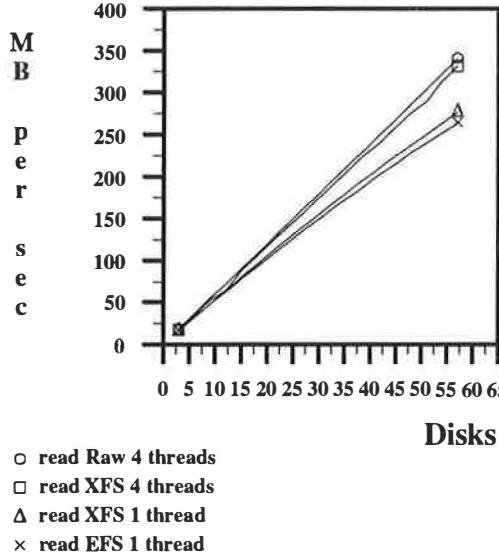


Figure 2. Read Throughput.

whereas the single threaded cases are not.

7.2. Database Sort Benchmark Results

Using XFS, Silicon Graphics recently achieved record breaking performance on the Datamation sort [Anon85] and Indy MinuteSort [Nyberg94] benchmarks. The Datamation sort benchmark measures how fast the system can sort 100 megabytes of 100 byte records. The MinuteSort benchmark measures how much data the system can sort in one minute. This includes start-up, reading the data in from disk, sorting it in memory, and writing the sorted data back out to disk. On a 12 CPU 200 Mhz Challenge system with 2.25 gigabytes of memory and a striped volume of 96 disk drives, we performed the Datamation sort in 3.52 seconds and sorted 1.6 gigabytes of data in 56 seconds for the MinuteSort. The previous records of 7 seconds and 1.08 gigabytes of data were achieved on a DEC Alpha system running VMS.

Achieving this level of results requires high memory bandwidth, high file system and I/O bandwidth, scalable multiprocessing, and a sophisticated multiprocessing sort package. The key contribution of XFS to these results is the ability to create and read files at 170 MB/sec. This actually moved the bottleneck in the system from the file system to the allocation of zeroed pages for the sort processes.

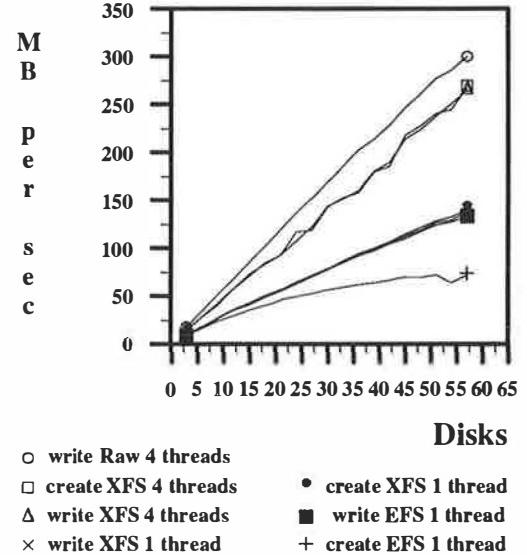


Figure 3. Write/Create Throughput.

7.3. LADDIS Benchmark Results

The results of the SPEC SFS (a.k.a. LADDIS) benchmark using XFS are encouraging as well. On a 12 CPU 250 Mhz Challenge XL with 1 gigabyte of memory, 4 FDDI networks, 16 scsi channels, and 121 disks, we achieved a maximum throughput of 8806 SPECnfs operations per second. While XFS plays only a part in achieving such outstanding performance, these results exceed our previous results using the EFS file system. On a slightly less powerful machine using EFS, we originally reported a result of 7023 SPECnfs operations per second. We estimate that the difference in hardware accounts for approximately 800 of the operations, leaving XFS approximately 1000 operations per second ahead of EFS. The difference is that EFS achieves 65 operations per second per disk, while XFS achieves 73 operations per disk. While this 12% increase might not seem like much, the LADDIS workload is dominated by small, synchronous write performance. This is often very difficult to improve without better disk hardware. We believe that the improvement with XFS is the result of the high performance directory structures, better file allocations, and synchronous metadata update batching of the transaction log provided by XFS.

7.4. Directory Lookups in Large Directories

Figure 4 contains the results for a test measuring the performance of random lookups in directories of various sizes for EFS and XFS. The results included are the average of several iterations of the test. The machine used for the test is a 4 CPU machine with 128 megabytes of memory. Each file system was created on a single, 2 gigabyte disk with nothing else on it. To make sure that we are measuring the performance of the file system directory structures, the test is run with the directory name lookup cache turned off. Also, the entries in the directories are all links to just a few real files. There are 20,000 links per real file. The test performs lookups using the stat(2) system call, so making most of the entries links to just a few files eliminates the size of the inode cache from the variability of the test.

Directory entries	EFS	XFS
100	5,970	8,972
500	2,833	6,600
1,000	1,596	7,089
10,000	169	6,716
30,000	43	6,522
50,000	27	6,497
70,000	-	5,661
90,000	-	5,497
150,000	-	177
250,000	-	102
350,000	-	90
450,000	-	79
1,000,000	-	66

Figure 4. Lookup Operations Per Second

It is clear from this test that lookups in medium to large directories are much more efficient using XFS. EFS uses a linear directory format similar to that used by BSD FFS. It degrades severely between 1,000 and 10,000 entries, at which point the test is entirely CPU bound scanning the cached file blocks for the entries being looked up. For XFS, the test is entirely CPU bound, but still very fast, until the size of the directory overflows the number of blocks that can be cached in memory. While there is a large amount of memory in the machine, only a limited portion of it can be used to cache directory blocks due to limitations of the IRIX metadata block cache. At the point where we overflow the cache, the interior nodes of the directory

B+ tree are still cached, but most leaf nodes in the tree need to be read in from disk when they are accessed. This reduces the performance of the test to the performance of directory block sized I/O operations to the single underlying disk drive. The reason the performance continues to degrade as the directory size increases is most likely that the effectiveness of the leaf block caching continues to decrease with the increase in directory size.

8. Conclusion

The main idea behind the design of XFS is very simple: think big. This idea brings forth the needs for large file systems, large files, large numbers of files, large directories, and large I/O that are addressed in the design and implementation of XFS. We believe that by satisfying these needs, XFS will satisfy the needs of the next generation of applications and systems so that we will not be back to where we are today in just a few years.

The mechanisms in XFS for satisfying the requirements of big systems also make it a high performance general purpose file system. The pervasive use of B+ trees throughout the file system reduces many of the algorithms in the file system from linear to logarithmic. The use of asynchronous transaction logging eliminates many of the metadata update performance problems in previous file systems. Also, the use of delayed allocation improves the performance of all file allocations, especially those of small files. XFS is designed to perform well on both the desktop and the server, and it is this focus on scalability that distinguishes XFS from the rest of the file system crowd.

9. Acknowledgments

We would like to thank John Ousterhout, Bob Gray, and Ray Chen for their help in reviewing and improving this paper; Chuck Bullis, Ray Chen, Tin Le, James Leong, Jim Orosz, Tom Phelan, and Supriya Wickrematillake, the other members of the XFS/XLV team, for helping to make XFS and XLV real, commercial products; and Larry McVoy for his magic troff incantations that made this paper presentable.

10. References

- [Anon85] Anonymous, "A Measure of Transaction Processing Power," Datamation, Vol. 31 No. 7, 112-118.
- [Baker91] Baker, M., Hartman, J., Kupfer, M., Shirriff, K., Ousterhout, J., "Measurements of a

- Distributed File System," Proceedings of the 13th Symposium on Operating System Principles, Pacific Grove, CA, October 1991, 192-212.
- [Chutani92] Chutani, S., Anderson, O., et. al., "The Episode File System," Proceedings of the 1992 Winter Usenix, San Francisco, CA, 1992, 43-60.
- [Comer79] Comer, D., "The Ubiquitous B-Tree," Computing Surveys, Vol. 11, No. 2, June 1979 121-137.
- [Dimino94] Dimino, L., Mediouni, R., Rengarajan, T., Rubino, M., Spiro, P., "Performance of DEC Rdb Version 6.0 on AXP Systems," Digital Technical Journal, Vol. 6, No. 1, Winter 1994 23-35.
- [Ganger94] Ganger, G., Patt, Y., "Metadata Update Performance in File Systems," Proceedings of the First Usenix Symposium on Operating System Design and Implementation, Monterey, CA, November, 1994, 49-60.
- [Hagmann87] Hagmann, R., "Reimplementing the Cedar File System Using Logging and Group Commit," Proceedings of the 10th Symposium on Operating System Principles, November, 1987.
- [Hisgen93] Hisgen, A., Birrell, A., Jerian, C., Mann, T., Swart, G., "New-Value Logging in the Echo Replicated File System," Research Report 104, Systems Research Center, Digital Equipment Corporation, 1993.
- [Hitz94] Hitz, D., Lau, J., Malcolm, M., "File System Design for an NFS File Server Appliance," Proceedings of the 1994 Winter Usenix, San Francisco, CA, 1994, 235-246.
- [Kleiman86] Kleiman, S., "Vnodes: an Architecture for Multiple File System types in Sun Unix," Proceedings of the 1986 Summer Usenix, Summer 1986.
- [McKusick84] McKusick, M., Joy, W., Leffler, S., Fabry, R. "A Fast File System for UNIX," ACM Transactions on Computer Systems Vol. 2, No. 3, August 1984, 181-197.
- [McVoy90] McVoy, L., Kleiman, S., "Extent-like Performance from a UNIX File System," Proceedings of the 1991 Winter Usenix, Dallas, Texas, June 1991, 33-43.
- [Nyberg94] Nyberg, C., Barclay, T., Cvetanovic, Z., Gray, J., Lomet, D., "AlphaSort: A RISC Machine Sort," Proceedings of the 1994 SIGMOD International Conference on Management of Data, Minneapolis, 1994.
- [Ousterhout85] Ousterhout, J., Da Costa, H., Harrison, D., Kunze, J., Kupfer, M., Thompson, J., "A Trace-Driven Analysis of the UNIX 4.2 BSD File System," Proceedings of the 10th Symposium on Operating System Principles, Orcas Island, WA, December 1985, 15-24.
- [Ousterhout90] Ousterhout, J. "Why Aren't Operating Systems Getting Faster As Fast as Hardware?" Proceedings of the 1990 Summer Usenix, Anaheim, CA, June, 1990, 247-256.
- [Rosenblum92] Rosenblum, M., Ousterhout, J., "The Design and Implementation of a Log-Structured File System," ACM Transactions on Computer Systems Vol 10, No. 1, February 1992, 26-52.
- [Sandberg85] Sandberg, R., et al., "Design and Implementation of the Sun Network File System," Proceedings of the 1985 Summer Usenix, June, 1985, 119-130.
- [Seltzer95] Seltzer, M., Smith, K., Balakrishnan, H., Chang, J., McMains, S., Padmanabhan, V., "File System Logging Versus Clustering: A Performance Comparison," Proceedings of the 1995 Usenix Technical Conference, January 1995, 249-264.
- [SGI92] IRIX Advanced Site and Server Administration Guide, Silicon Graphics, Inc., chapter 8, 241-288
- [Veritas95] Veritas Software, <http://www.veritas.com>
- Adam Sweeney, Doug Doucette, Wei Hu, Curtis Anderson, Michael Nishimoto, and Geoff Peck** are all members of the Server Technology group at Silicon Graphics. Adam went to Stanford, Doug to NYU and Berkeley, Wei to MIT, Curtis to Cal Poly, Michael to Berkeley and Stanford, and Geoff to Harvard and Berkeley. None of them holds a Ph.D. All together they have worked at somewhere around 27 companies, on projects including secure operating systems, distributed operating systems, fault tolerant systems, and plain old Unix systems. None of them intends to make a career out of building file systems, but they all enjoyed building one.

A Comparison of FFS Disk Allocation Policies

Keith A. Smith and Margo Seltzer

Harvard University

Abstract

The 4.4BSD file system includes a new algorithm for allocating disk blocks to files. The goal of this algorithm is to improve file clustering, increasing the amount of sequential I/O when reading or writing files, thereby improving file system performance. In this paper we study the effectiveness of this algorithm at reducing file system fragmentation. We have created a program that artificially ages a file system by replaying a workload similar to that experienced by a real file system. We used this program to evaluate the effectiveness of the new disk allocation algorithm by replaying ten months of activity on two file systems that differed only in the disk allocation algorithms that they used. At the end of the ten month simulation, the file system using the new allocation algorithm had approximately half the fragmentation of a similarly aged file system that used the traditional disk allocation algorithm. Measuring the performance difference between the two file systems by reading and writing the same set of files on the two systems showed that this decrease in fragmentation improved file write throughput by 20% and read throughput by 32%. In certain test cases, the new allocation algorithm provided a performance improvement of greater than 50%.

1 Introduction

Recent file systems [Peacock88][McVoy90] have used *clustering* to improve performance. These systems attempt to place logically sequential file data on physically contiguous disk blocks. When such layout is achieved, large multiple block transfers can be used to read/write files at close to the disk system's maximum bandwidth. Measurements have shown that these clustering enhancements can improve performance by a factor of two or three [McVoy90] [Seltzer93] over file systems that perform I/O one block at a time. These performance measurements

were made on empty file systems and represent the best case behavior for clustering file systems. Over time, clustering is less successful, because free space becomes fragmented and the disk allocation algorithms fail to fully exploit existing free clusters. A recent study showed that UNIX file systems that are more than two years old perform as much as 15% worse than comparable empty file systems [Seltzer95]. This decline in performance correlates closely with increased fragmentation of newly created files on these file systems. Although the maximum transfer size on these file systems was seven file system blocks, the average cluster size for mid-sized files (32–128KB) was only three blocks.

As file systems age, clustered file allocation becomes less successful because the file system is unable to find clusters of free space from which to make allocations. This may occur either because free space becomes too fragmented to support clustering, or because the file system does not fully exploit existing clusters of free space when allocating space for new files. An examination of the file systems on several file servers at Harvard showed that there are many large clusters of free space on UNIX file systems that are two to three years old [Smith94]. We conclude from this that the file fragmentation observed on real file systems is the result of a disk allocation algorithm that is unable to find and exploit existing clusters of free space.

In the hopes of providing better long term clustering, Kirk McKusick modified the disk allocation algorithm used by the 4.4BSD-Lite Fast File System (FFS) to better exploit existing clusters of free space [CSRG94].

In order to understand the long term effectiveness of this new allocation algorithm, we have developed a tool that simulates a ten month work load in order to artificially age a file system. We used this tool to age two different file systems, one that used the original disk allocation algorithm and one that used the new allocation algorithm. By tracking the amount of file fragmentation during the course of the simulation, we compared the effectiveness of the two disk algorithms. We also compared the performance of the resulting aged file systems to understand the impact of the resulting differences in fragmentation.

This research was supported by Sun Microsystems Laboratories and by the National Science Foundation under grant CCR-9502156.

In Section 2, we describe the disk allocation algorithm that has traditionally been used by the UNIX Fast File System and explain the improvements offered by the new algorithm. Section 3 explains the file system aging process, including the methodology used to generate the aging workload and a validation of the aging program by comparing its results to a real file system. In Section 4, we compare the file fragmentation that results from the original FFS allocation algorithm and the new allocation algorithm. Section 5 provides a performance comparison of the aged file systems. Section 6 discusses some future research directions based on the results of this work. Section 7 summarizes this study.

2 FFS Disk Allocation Algorithms

A simplified explanation of the original FFS disk allocation algorithm is presented here. A more detailed explanation may be found in *The Design and Implementation of the 4.3BSD UNIX Operating System* [Leffler89].

FFS divides the disk into *cylinder groups*, each of which is a set of consecutive cylinders. Cylinder groups are used to exploit locality; related data are co-located in the same cylinder group. Thus FFS allocates logically sequential blocks of a file in the same cylinder group, and likewise allocates all of the files in a directory to the same cylinder group as the directory.

The FFS disk allocation policy is divided into two steps. When a new block is allocated to a file, FFS first determines the cylinder group from which the block will be allocated. FFS then selects a free block from that cylinder group and allocates it. Selecting a cylinder group is a simple task; FFS uses the cylinder group where the previous block(s) of the file are located¹. In this paper, we focus on the second part of the allocation—selecting a block from within a cylinder group.

The original FFS disk allocation algorithm allocates one block at a time to a file, attempting to allocate contiguous blocks where possible. When a new block is allocated, FFS determines the location of the previous block of the file and attempts to allocate the next disk block. If this block is not available, FFS allocates a different block from the cylinder group, attempting to find one that minimizes the seek time from the previous block of the file. The selection of

1. To prevent one large file from filling an entire cylinder group, each time an indirect block is allocated to a file, allocation changes to a different cylinder group.

this alternate block does *not* consider the amount of free space where the new block is located. Thus if there is just one free block in a good location and a cluster of ten free blocks in a slightly worse location, FFS will allocate the single free block, making it impossible to perform contiguous allocation after that block.

McKusick's new allocation algorithm adds a reallocation step to the original FFS disk allocation algorithm. For this reason it is referred to as the *realloc* algorithm. FFS initially allocates blocks in the manner described above. Before the blocks are written to disk, however, the reallocation code gathers clusters of logically sequential blocks and tries to relocate them to free clusters of the appropriate size. The maximum size cluster produced by the realloc code is determined by a file system parameter, and is usually configured to be the maximum I/O transfer size of the underlying disk system.

3 File System Aging

To understand the effectiveness of a disk allocation algorithm, the long term effect of the algorithm on file system layout must be examined. To do this in a laboratory setting, we generated an artificial load intended to simulate the pattern of file operations that a file system sees over an extended period of time. This process is called *file system aging*. After aging a file system, the layout of its files can be analyzed and compared with similarly aged file systems that used different allocation algorithms.

3.1 Generating a Workload

The central problem in aging a file system is generating a realistic workload. Because a test system is likely to start with an empty disk, this workload should start with an empty file system and simulate the load on a new file system over many months or years. The ideal method for generating this workload would be to collect extended file system traces and to age a test file system by replaying the exact set of file operations seen in the trace. The duration of the required traces makes this strategy impractical. Instead, we generated a workload from two sets of file system data that were readily available. Fortunately, an exact reproduction of the load on a real file system is not required.

We used a set of file system *snapshots* collected from a file system on a local file server to simulate the day-to-day changes on a file system. These snapshots, which were originally collected for a study of file system fragmentation [Smith94], were collected

nightly over a period of one year. Each snapshot describes all of the files on a file system at the time of the snapshot. For each file the snapshot includes the file's inode number, inode change time, file type, file size, and a list of the disk blocks allocated to the file.

By comparing successive snapshots of one file system, we generated a list of the files that were created, deleted, or modified on each day. In order to simulate the activity on an empty file system, we chose a file system that was nearly empty at one point in the snapshot collection period, using the point of lowest utilization (9% full) as the starting time.

The major obstacle to accurately reproducing the original workload from the file system snapshots was interpolating the file system activity that occurred between successive snapshots. By comparing the list of allocated inodes in two snapshots, it was easy to determine which files were created, deleted, or modified during the intervening interval. Unfortunately, the snapshots did not provide sufficient information to determine the exact time at which these operations took place.

We used several heuristics to assign creation and deletion times to these file operations. Previous studies have shown that files are seldom modified after they are first written [Ousterhout85]. Therefore, when a new file was created, we used its inode change time as the time the file was created. Similarly, if a file was modified, we assumed that it had been removed (or truncated to zero length) and then rewritten. The most difficult operations to which to assign times were deletes. When a file was deleted between two snapshots, there was no information that provided hints about the time it was deleted. We randomly assigned times to file deletions, making sure that they fell during the range of times that other operations were occurring on the file system.

Another difficulty in recreating the file system's workload from the daily snapshots was accounting for files that were created and then deleted on the same day. Trace-based file system studies [Ousterhout85] [Baker91] have shown that most files live for less than the twenty-four hours between successive snapshots. These files, which did not show up in the snapshots, can affect the fragmentation of the longer-lived files on the file system. To approximate the additional file creations and deletions generated by these short-lived files, we used multiple-day traces of NFS requests to Network Appliance file servers [Hitz94]. This data, which was originally used in a study of cleaning algorithms for log-structured file systems [Blackwell95], includes all of the create, delete, and write requests issued to the servers during the trace periods.

We generated a list of all of the files created and deleted during each 24-hour period in the NFS traces. These files were sorted by the day they were created and the directory in which they were created (the directory information is available in the create requests). The result was a trace log describing all of the files that were created and then deleted on the same day.

The next step was to integrate these short-lived files into the workload generated from the file system snapshots. For each day in the snapshot period, we randomly selected one day from the NFS traces, and integrated that day's short-lived file activity into the aging workload. The file operations from the NFS traces were placed in the directories that had the most changes between snapshots. To ensure that the NFS operations overlapped with the file operations generated from the snapshots, all of the NFS operations in each directory were time-shifted to coincide with the peak period of activity in the directory to which they were added. The end result was that the operations on short-lived files (generated from the NFS traces) were interleaved with the creations and deletions of longer-lived files (generated from the snapshots).

The resulting workload simulates ten months of activity (from April, 1994 through February, 1995) on a 502 megabyte file system. The source file system is used for the home directories of one professor and three students in a networking research group. At the beginning of the ten month period, the file system was 9% utilized, and for most of the ten month period utilization was greater than 70%, reaching a high of 90%². The workload contains approximately 800,000 file operations that write 48.6 gigabytes of data to the disk and take fourteen hours to replay on our test machine.

3.2 Replaying the Workload

To age a file system, we applied the workload described above to an empty file system. The aging program reads records from the workload file, performing the specified file operations. This task was complicated by the fact that complete pathnames for the created files were not available in the snapshots used to generate the workload. Because FFS assigns files to cylinder groups based on the cylinder group of the file's directory, the algorithm used by the aging

2. These utilization numbers treat FFS's free space reserve (10% of the disk) as free space.

CPU Parameters		Disk Parameters		File System Parameters	
CPU	Intel Pentium	Disk Controller	Bustek 946C (SCSI)	Size	502 MB
Clock Speed	120 MHz	Disk Type	Seagate 32430N	Fragment Size	1 KB
Memory	64 MB	Total Disk Space	2.1 GB	Block Size	8 KB
Bus Type	PCI	Rotational Speed	5411 RPM	Max. Cluster Size	56 KB
		Sector Size	512 Bytes	Rotational Gap	0
		Cylinders	3992	Cylinder Groups	27
		Heads	9	Heads	22
		Average Sectors per Track	116	<i>Sectors per Track</i>	<i>118</i>
		Track Buffer	512 KB		
		Average Seek	11 ms		

Table 1: Benchmark Configuration. This table describes the hardware configuration used for benchmarking and for verifying the file system aging workload. The file system parameters shown in italics were set to match the file system from which the aging workload was generated despite the fact that they do not match the underlying hardware.

program to assign files to directories can have a major impact on the accuracy of the aging simulation.

In the absence of the original pathnames in the file system snapshots, we decided to simply create the files in the correct cylinder groups. Cylinder groups represent the pools from which disk blocks are allocated. By creating files in the same cylinder group on the simulated file system as on the original file system, we ensured that each cylinder group on the simulated file system saw the same set of allocation and deallocation requests that were presented to the corresponding cylinder group on the original file system. We used each file's inode number to compute the cylinder group to which it was allocated on the original file system. To force the files into the same cylinder groups on the aged file system, we exploited several details of the FFS implementation.

We started the aging process with an empty file system. The first step was to create one directory for each cylinder group on the file system. The algorithm used by FFS to assign directories to cylinder groups ensures that each directory was placed in a different cylinder group. For each file in the aging workload, we used its inode number to compute the cylinder group to which it was allocated on the original file system, and placed the file in the corresponding directory on the aged file system. Because FFS places all files in the same cylinder group as their directory, this guaranteed that all of the files that were in the same cylinder group on the original file system were also in the same cylinder group on the aged file system. Thus, the sequence of block allocation and

freeing operations in each cylinder group was the same as on the original file system.

This approach does have one minor disadvantage. By creating an extra directory for each cylinder group, we are introducing one file per cylinder group that did not exist in any of the data sets used to generate the aging workload (i.e., the directory). The effect of these extra directories is negligible, however, since the space they occupy (approximately 300 kilobytes) represents less than 0.1% of the total disk utilization during the aging simulation.

3.3 Verifying the Aging Process

To verify the accuracy of the aging process, we compared the file fragmentation on an artificially aged file system with the fragmentation on the original file system that was used to generate the aging workload. We define a *layout score* to quantify the amount of file fragmentation in a file or file system. The layout score for an individual file is the fraction of that file's blocks that are optimally allocated. An optimally allocated block is one that is physically contiguous with the previous block of the same file. The first block of a file is not included in this calculation, since it is impossible for it to have a "previous block." Similarly, layout score is undefined for one block files, since they cannot be fragmented. A file with a layout score 1.00 is perfectly allocated; all of its blocks are contiguously allocated. A file with a layout score of 0.00 has no contiguously allocated blocks.

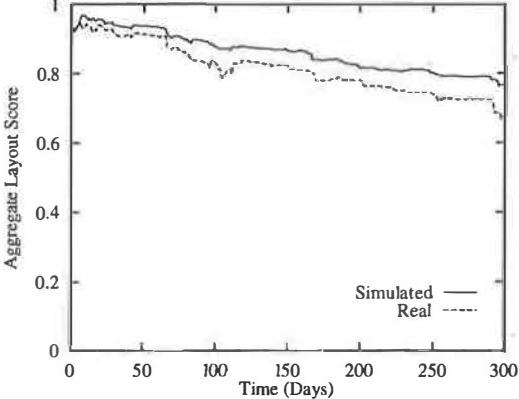


Figure 1: Aggregate Layout Score Over Time: Real vs. Simulated File Systems. This chart plots the aggregate layout score for each day in the ten month simulation period. The “Simulated” line shows the fragmentation on the artificially aged file system. The “Real” line shows the fragmentation on the original file system from which the aging workload was generated.

To evaluate the fragmentation of all of the files on a file system, we compute the file system’s *aggregate layout score*. This metric is the fraction of the file system’s allocated blocks that are optimally allocated (again ignoring the first block of each file and one block files).

To verify the accuracy of the aging process, we constructed a file system with the same parameters as the file system from which the aging workload was generated. These parameters, along with our hardware configuration, are summarized in Table 1. We used BSD/OS Version 2.0.1 for these and all of the other measurements in this paper. We then ran the aging workload on this file system, computing the file system’s aggregate layout score at the end of each simulated day in the workload. For comparison, we also computed the aggregate layout score on the original file system for each day during the period from which the aging workload was generated. The resulting layout scores for the two file systems are plotted in Figure 1.

The simulated file system has higher layout scores than the original file system, indicating that the aging process does not cause as much file fragmentation as the original file system experienced. At the end of the ten month period, the simulated file system’s aggregate layout score was 0.77, compared to the 0.68 aggregate layout score of the original file system. Despite the greater fragmentation on the original file system, the two file systems exhibit comparable behavior; as can be seen by the similar contours of the two curves in Figure 1; the simulated file system has many of the same drops and jumps as the original, although they are of smaller magnitude.

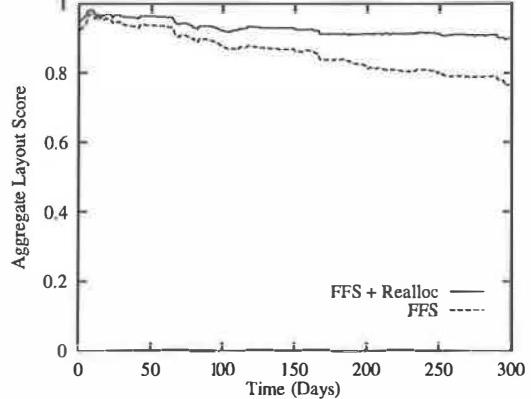


Figure 2: Aggregate Layout Score Over Time: FFS vs. realloc algorithm. This chart plots the aggregate layout score on each day of the ten month simulation period. The “FFS” line shows the aggregate layout scores on the file system that used the original FFS disk allocation algorithm. The “FFS + Realloc” line shows the aggregate layout scores on the file system that used the new realloc allocation algorithm.

There are also some areas where the simulated file system has failed to capture changes that occurred on the original. The clearest example of this is in days 90–110 of the simulation, where the aggregate layout score of the original file system changes from day to day while the layout score of the simulated file system remains relatively constant.

The difference in fragmentation between the two file systems is the result inaccuracies in the aging workload. Section 3.1 discussed the approximations that were necessary due to the incomplete information contained in the file system snapshots used to generate the workload. Despite these differences, the aging workload is realistic in many ways. As on the real file system, the layout score of the artificially aged file system decreases steadily over time, occasionally declining quickly for a day or two, sometimes remaining almost constant for many weeks.

4 Comparison of Allocation Algorithms

To compare the FFS disk allocation algorithm to the realloc algorithm, we aged two file systems that differed only in the disk allocation algorithm that they used. These tests were also performed on the hardware configuration described in Table 1. After each simulated day during the aging, we computed the aggregate layout score for the two file systems. The results are plotted in Figure 2.

The file system that used the realloc allocation algorithm exhibited less fragmentation (higher layout scores) for the entire duration of the 300 day

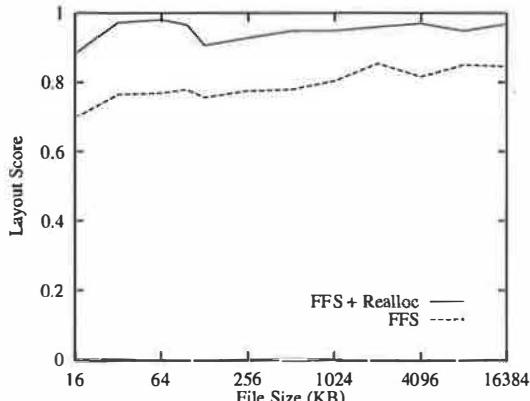


Figure 3: Layout Score as a Function of File Size. The “FFS” line was generated from the aged file system that used the original FFS allocation algorithm. The “FFS + Realloc” was generated from the aged file system that used the realloc enhancements.

simulation. The difference in aggregate layout score between the two file systems increased over time, from a difference of 0.026 (0.950 vs. 0.924) after the first day of the simulation, to a difference of 0.133 (0.899 vs. 0.766) at the end of the simulation. In other words, by the end of the simulation only 10.1% of the file blocks were non-optimally allocated when using the realloc algorithm, in contrast to 23.4% when the realloc code was not used—an improvement of 56.8%.

To understand the types of files that derive the most benefit from the realloc algorithm, we sorted the files on both file systems by size, and computed the aggregate layout score for files of a variety of sizes. The results are shown in Figure 3. This graph shows that the realloc disk allocation algorithm produces better file layout (i.e., less fragmentation) for all file sizes, and near optimal layout for files smaller than the file system cluster size. Surprisingly, two block files have a lower layout score than slightly larger files when the realloc algorithm is used. This is due to a quirk in the disk allocation code, which does not invoke the realloc functionality until a file fills the second block. The lower layout score for two block files in Figure 3 is the result of files that are big enough to use two blocks (instead of one block and some number of fragments) but do not completely fill the second block.

Both file systems in Figure 3 exhibit a drop in layout score when file size passes twelve blocks (96 KB). Files larger than twelve blocks require an indirect block, which is always allocated in a different cylinder group than the first part of the file. The result is that all files of more than twelve blocks contain at least one non-optimal block (the thirteenth), lowering

the average layout score for these files. As the file size grows past thirteen blocks, the effect of this mandatory seek becomes smaller, and the layout score rises again.

5 Performance

We expected the decreased fragmentation seen when using the realloc algorithm to lead to better file system performance. In this section we present the results of several benchmarks that quantify the performance difference between file systems using the two allocation algorithms.

The most important difference in the two disk allocation algorithms is the long-term effect that they have on file layout. To account for this in benchmarking the two FFS implementations, we ran all of our benchmarks on file systems that were aged using the ten month aging workload described in Section 3.

One set of benchmarks measures the performance of sequential reads and writes to files of varying sizes. A second benchmark uses the files left on the file systems at the end of the aging process to compare the performance of files that were created in a more realistic manner.

5.1 Sequential I/O Performance

Our first measurement compared the file systems using a benchmark of sequential read and write performance. The benchmark operated on thirty-two megabytes of data, which was decomposed into the appropriate number of files for the file size being measured. Because FFS allocates all of the files in a single directory to the same cylinder group, the data was divided into subdirectories, each containing no more than twenty-five files. This distributed the benchmark data across more cylinder groups than if all of the test files had been placed in one directory.

The benchmark executed in two phases:

1. **Create/write:** All of the files were created. For file sizes of four megabytes or smaller, the entire file was created with one write operation. Large files were created using as many four megabyte writes as necessary.
2. **Read:** The files were read in the same order in which they were created. As with the create phase, I/O was performed in four megabyte units.

We ran this benchmark for a variety of file sizes from sixteen kilobytes (two file blocks) to thirty-two

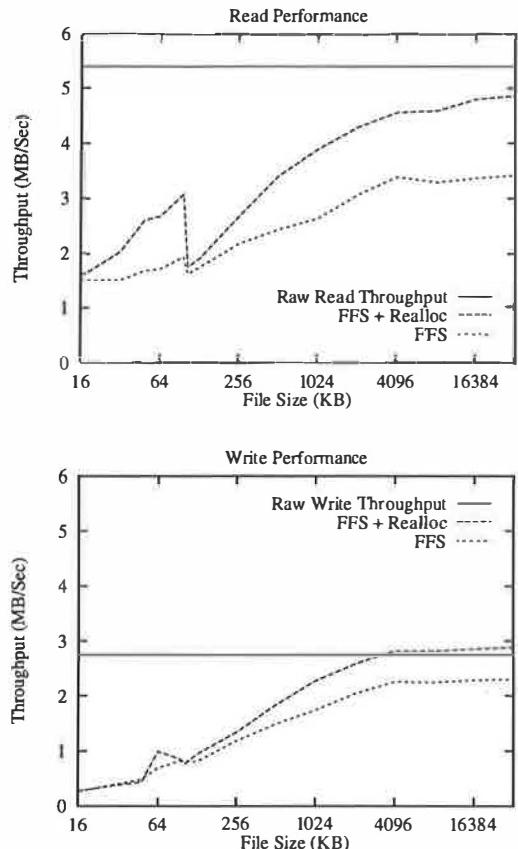


Figure 4: Sequential I/O Performance. These graphs show the read and write performance of the sequential I/O benchmark on the two FFS implementations. The throughput reading and writing the raw disk are also shown. In the top graph, showing read performance, we see that when the realloc algorithm is used, performance improves by as much as 58%. Similarly, as shown in the bottom graph, the realloc algorithm improves write performance by as much as 44%. For large file sizes, performance using the realloc algorithm exceeds the throughput to the raw disk. This surprising result is due to lost rotations when writing the raw disk. All benchmarks were executed ten times and had standard deviations smaller than 1.5% of the mean data value.

megabytes. The results, presented in Figure 4, show that the FFS implementation that included the realloc algorithm performed better for nearly all file sizes. The sharp dip in all of the performance curves at 104 KB corresponds to the file size at which FFS begins to use indirect blocks. Because FFS allocates indirect blocks, and the data blocks to which they point, in a different cylinder group than the previous part of the file, a large performance penalty is incurred at this file size. The overhead of this seek between cylinder groups is amortized as the file size grows, improving throughput for larger file sizes.

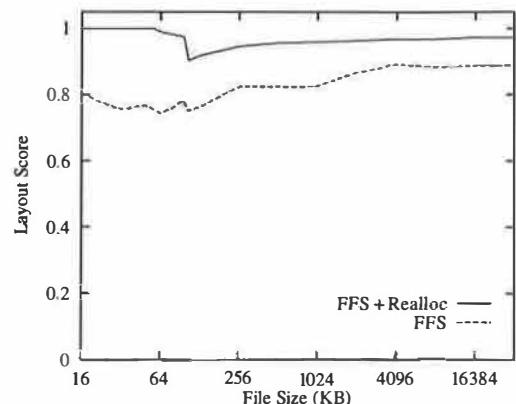


Figure 5: File Fragmentation During Sequential I/O Benchmark. This graph shows the average layout score of files created by the sequential I/O benchmark as a function of file size. The ‘FFS’ line was generated from the aged file system using the original FFS disk allocation algorithm. The ‘FFS + Realloc’ line was generated from the aged file system using the realloc allocation algorithm. For all file sizes, the realloc algorithm produced better file layout. For files up to fifty-six kilobytes (the file system cluster size) the realloc algorithm achieved perfectly contiguous file layout.

Small file reads exhibit a large performance difference between the two FFS implementations. 96 kilobyte files, the largest size possible without an indirect block, have 58% greater read throughput on the file system with the realloc disk allocation algorithm. This performance improvement is directly attributable to the better layout attained when using the realloc algorithm. Figure 5, which graphs the average layout score of the files created for each run of the benchmark, shows that for files of up to fifty-six kilobytes (the file system cluster size) the realloc algorithm attained perfectly contiguous file layout.

The improvement in create performance when using the realloc algorithm is less noticeable than the corresponding change in read performance, especially for smaller file sizes. This smaller performance difference is due to the synchronous metadata updates that FFS performs when creating a file. These metadata updates dominate the total run time of the create benchmark, and differences in file layout have little effect on the performance of small file creates.

For larger files, there was a more noticeable improvement in write performance when using the realloc algorithm. Large files (four megabytes and larger) perform 25% better, and 64 kilobyte files perform 44% better using the realloc algorithm than they do on the original FFS.

It is interesting to note that write performance when using the realloc algorithm drops after 64 kilobytes, unlike read performance which does not

drop off until the first indirect block is allocated at 104 kilobytes. This is an artifact of the maximum disk transfer size imposed by the hardware (64 kilobytes). As Figure 5 indicates, most files between 64 and 96 kilobytes are allocated completely contiguously, despite the fact that they require more than one cluster on the disk. When writing to such a file, the first 64 kilobytes of data is transferred in one request, and the remaining data in a second request. By the time the second request has been issued, however, the disk has rotated past the location where the data is to be written, adding the latency of an extra disk rotation to the I/O time. This phenomenon does not occur when reading the same file because of the read-ahead performed by the track buffer [Seltzer95].

These lost disk rotations between sequential write requests also explain why the write throughput to large files when using the realloc algorithm exceeds the write throughput to the raw disk. When writing to the raw disk, all writes are sequential, and a rotation is lost between each transfer. When the realloc algorithm is used, large files achieve good, but not perfect layout (as shown in Figure 5). These imperfections actually improve write performance, as a small seek between transfers is preferable to a lost rotation. A similar benefit is not seen for large files using the original FFS allocation algorithm because the resulting layout is more fragmented. The overhead of these additional seeks exceeds the savings from avoiding extra rotations.

The performance improvement seen when using the realloc algorithm was larger than we had anticipated. Before running the sequential I/O benchmark, we had expected to see performance differences of no more than 15%, in line with previous research comparing the performance of contiguous and fragmented FFS files [Seltzer95]. The larger than expected performance improvements seen in our tests of the realloc algorithm are explained by comparing our hardware configuration to the one used in the earlier research. Although the two systems had comparable disks, the SparcStation 1 used in the earlier study provided substantially less I/O bandwidth than the PCI bus in our current test configuration. As a result, the ratio of seek time to transfer time was higher on the PCI-based system, and reducing the seek time resulted in larger performance improvements (expressed as a percentage of the total I/O time) than were possible on the SparcStation.

	FFS	FFS + Realloc
Layout Score	0.80	0.96
Read Throughput	1.65 MB/sec	2.18 MB/sec
Write Throughput	1.04 MB/sec	1.25 MB/sec

Table 2: Performance of Recently Modified Files.

This table presents the read and write throughput of the files modified during the last month of the aging simulation. The aggregate layout score of these files is also presented. The “FFS” column provides the measurements on a standard FFS file system. The “FFS + Realloc” column presents the same measurements on an FFS that includes the realloc disk allocation algorithm. Each of the throughput tests was run ten times. All standard deviations were less than 2% of the corresponding mean value.

5.2 Existing File Performance

One important aspect of the previous benchmark is not representative of many types of real world file system usage. On a real file system, files are deleted as well as created; this may create small holes of free space that cause fragmentation in subsequently created file. Unlike the sequential I/O benchmark, the aging workload interleaves many create and delete operations, possibly resulting in more file fragmentation than is produced by the sequential I/O benchmark. To gauge the effect of using more “realistically” created files, we ran some benchmarks using the files that were present on the test file systems at the end of the aging process.

Previous research has shown that most older files are seldom accessed [Satyanarayanan81], and therefore that the most active files on a file system tend to be relatively young. We approximated the set of “hot” files on our simulated file system by using all of the files that were modified during the last month of the aging workload. These files represent 10.5% of the files on the aged file system (929 out of 8774 files), and use 59.5 megabytes of storage (19% of the allocated disk space). Since these files cannot all fit in the buffer cache, their layout and performance should have a large effect on the overall performance of the file system.

We measured the file system throughput when reading and writing this complete set of files. To limit the amount of time spent seeking between files, we sorted the files by directory, so multiple files would be read from one cylinder group before moving to another. In order to preserve the layout of the original

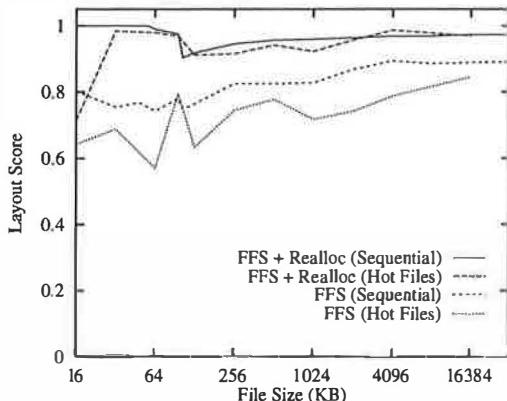


Figure 6: Layout Score of Hot Files. This graph plots the layout score of the hot file set on the two file systems as a function of file size. For comparison, the layout scores of the files produced by the sequential I/O benchmarks (from Figure 5) are also graphed.

files, the write phase of this benchmark overwrote the existing files. Thus the write performance does not include the overhead of creating the files or of allocating disk space to them (these overheads were included in the sequential write performance measurements of Section 5.1). We also computed the aggregate layout score of the files used in this test. The results, presented in Table 2, show that the FFS with the realloc algorithm had 32% higher read throughput and 20% higher write throughput than the original FFS. The performance difference between the two file systems on this benchmark is consistent with the sequential I/O performance measurements shown in Figure 4.

Figure 6 shows the layout score of the hot files plotted as a function of file size for the two file systems. For comparison, we also present the same data for the files from the sequential I/O test (copied from Figure 5). These data show that although the sequential I/O tests produced better layout than the “hot” files under the original FFS implementation, with the realloc algorithm, the layout of the hot files is almost identical to that of the files produced by the sequential I/O test. This indicates the ability of the realloc algorithm to produce near optimal file layout in a variety of circumstances. In the hot file benchmark on the file system that used the realloc algorithm, the layout score for two block files is lower than any other layout score measured in this test. The poor layout of two block files is a result of the same behavior described in Section 4.

6 Future Work

We believe that file system aging can be used to address two issues frequently overlooked in file

system performance analysis. This first is the simple fact that real-world file systems usually operate at close to full utilization, unlike the empty file systems that are often used when analyzing file system behavior in a laboratory setting. The second issue that file system aging allows us to address is the impact of specific design decisions on the long term behavior of a file system. The interaction of the disk allocation algorithm and file layout examined in this paper is one such issue. There are a variety of other issues in FFS and other file systems that readily suggest themselves for similar analysis.

In order to apply the file system aging technique to other file systems, we need to generalize the way the aging workload is replayed. The current program makes an important assumption about the behavior of the underlying file system in the way it assigns file operations to directories. More work also needs to be done to make the aging program work on file systems where the idle time between file operations can effect the behavior of the file system itself. An example of this is the timing of cleaner execution on a log-structured file system [Rosenblum92].

We also plan to generate a variety of different aging workloads representative of different file system usage patterns, such as news, database, and personal computing workloads. By analyzing the demands of different workloads, we hope to determine the file system design parameters that are best suited for each type of workload.

7 Conclusions

Our simulations and benchmarks provide conclusive evidence of the improved file layout and file system performance achieved using the realloc disk allocation algorithm. A simulation of ten months of file system activity shows that the reallocation algorithm decreases the number of intra-file disk seeks by more than 50%. With the exception of the mandatory seek imposed by FFS when a file becomes large enough to require an indirect block, the realloc algorithm produces nearly optimal file layout.

In all of the benchmarks that we conducted, an FFS using the realloc code outperformed a file system that did not include this enhancement. The improved file layout achieved by the realloc algorithm improved read and write performance for large files by up to 16%. Read performance for files up to 96 kilobytes improved by as much as 20%. The synchronous metadata updates performed when creating a file limited the performance improvements for writing small files.

8 Availability

The source code for the aging tool and the benchmarks along with the aging workload are available on the World Wide Web at:

<http://www.eecs.harvard.edu/~keith>

9 Acknowledgments

We would like to express our gratitude to the many reviewers for their useful guidance and suggestions, and to offer special thanks to Christopher Small and Jackie Horne for their last minute proof-reading.

10 References

- [Bach86] Bach, M., *The Design and Implementation of the UNIX Operating System*, Prentice-Hall, Englewood Cliffs, NJ, 1986.
- [Baker91] Baker, M., Hartman, J., Kupfer, M., Shirriff, K., Ousterhout, J., "Measurements of a Distributed File System," *Proceedings of the Thirteenth Symposium on Operating Systems Principles*, Monterey, CA, October 1991, pp. 198–212.
- [Blackwell95] Blackwell, T., Harris, J., Seltzer, M., "Heuristic Cleaning Algorithms in Log-Structured File Systems," *Proceedings of the 1995 USENIX Technical Conference*, New Orleans, LA, January 1995, pp. 277–288.
- [CSRG94] Computer Systems Research Group, University of California at Berkeley, *4.4BSD-Lite Source CD-ROM*, USENIX Association and O'Reilly & Associates, Sebastopol, CA, 1994.
- [Hitz94] Hitz, D., Lau, J., Malcolm, M., "File System Design for an NFS File Server Appliance," *Proceedings of the Winter 1994 USENIX Conference*, San Francisco, CA, January 1994, pp. 235–246.
- [Leffler89] Leffler, S., McKusick, M., Karels, M., Quarterman, J., *The Design and Implementation of the 4.3BSD UNIX Operating System*, Addison-Wesley Publishing Company, Reading, MA, 1989.
- [McKusick84] McKusick, M., Joy, W., Leffler, S., Fabry, R., "A Fast File System for UNIX," *ACM Transactions on Computer Systems*, Vol. 2, No. 3, August 1984, pp. 181–197.
- [McVoy90] McVoy, L., Kleiman, S., "Extent-like Performance from a UNIX File System," *Proceedings of the Summer 1990 USENIX Conference*, Anaheim, CA, June 1990, pp. 137–144.
- [Ousterhout85] Ousterhout, J., Costa, H., Harrison, D., Kunze, J., Kupfer, M., Thompson, J., "A Trace-Driven Analysis of the UNIX 4.2BSD File System," *Proceedings of the Tenth Symposium on Operating Systems Principles*, Orcas Island, WA, December 1985, pp. 15–24.
- [Peacock88] Peacock, J., "The Counterpoint Fast File System," *Proceedings of the Winter 1988 USENIX Conference*, Dallas, TX, February 1988, pp. 243–249.
- [Rosenblum92] Rosenblum, M., Ousterhout, J., "The Design and Implementation of a Log-Structured File System," *ACM Transactions on Computer Systems*, Vol. 10, No. 1, February 1992, pp. 26–52.
- [Satyanarayanan81] Satyanarayanan, M., "A Study of File Sizes and Functional Lifetimes," *Proceedings of the Eighth Symposium on Operating Systems Principles*, Pacific Grove, CA, December 1981, pp. 96–108.
- [Seltzer93] Seltzer, M., Bostic, K., McKusick, M., Staelin, C., "The Design and Implementation of the 4.4BSD Log-Structured File System," *Proceedings of the Winter 1993 USENIX Conference*, San Diego, CA, January 1993.
- [Seltzer95] Seltzer, M., Smith, K., Balakrishnan, H., Chang, J., McMains, S., Padmanabhan, V., "File System Logging Versus Clustering: A Performance Comparison," *Proceedings of the 1995 USENIX Technical Conference*, New Orleans, LA, January 1995, pp. 249–264.
- [Smith94] Smith, K., Seltzer, M., "File Layout and File System Performance," Harvard University Computer Science Department Technical Report TR-35-94.

Keith A. Smith (keith@eecs.harvard.edu) is a graduate student in computer science at Harvard University. His research interests include file system design and performance, extensible operating systems, and the search for the perfect pizza recipe. Keith received his B.S. in computer science from Yale University in

1987. Prior to entering the Ph.D. program at Harvard, Keith worked for VenturCom, Inc., developing a real-time UNIX for the x86 architecture.

Margo I. Seltzer (margo@eecs.harvard.edu) is an Assistant Professor of Computer Science in the Division of Applied Sciences at Harvard University. Her research interests include file systems, databases, and transaction processing systems. She is the co-author of several widely-used software packages including database and transaction libraries and the 4.4BSD log-structured file system. Dr. Seltzer spent several years working at start-up companies designing and implementing file systems and transaction processing software and designing microprocessors. She is a Sloan Foundation Fellow in Computer Science and was the recipient of the University of California Microelectronics Scholarship, the Radcliffe College Elizabeth Cary Agassiz Scholarship, and the John Harvard Scholarship. Dr. Seltzer received an A.B. degree in Applied Mathematics from Harvard/Radcliffe College in 1983 and a Ph. D. in Computer Science from the University of California, Berkeley in 1992.

AFRAID — A Frequently Redundant Array of Independent Disks

Stefan Savage
University of Washington, Seattle, WA

John Wilkes
Hewlett-Packard Laboratories, Palo Alto, CA

Abstract

Disk arrays are commonly designed to ensure that stored data will *always* be able to withstand a disk failure, but meeting this goal comes at a significant cost in performance. We show that this is unnecessary. By trading away a fraction of the enormous reliability provided by disk arrays, it is possible to achieve performance that is almost as good as a non-parity-protected set of disks.

In particular, our AFRAID design eliminates the small-update penalty that plagues traditional RAID 5 disk arrays. It does this by applying the data update immediately, but delaying the parity update to the next quiet period between bursts of client activity. That is, AFRAID makes sure that the array is *frequently* redundant, even if it isn't always so. By regulating the parity update policy, AFRAID allows a smooth trade-off between performance and availability.

Under real-life workloads, the AFRAID design can provide close to the full performance of an array of unprotected disks, and data availability comparable to a traditional RAID 5. Our results show that AFRAID offers 42% better performance for only 10% less availability, 97% better for 23% less, and as much as a factor of 4.1 times better performance for giving up less than half RAID 5's availability.

We explore here the detailed availability and performance implications of the AFRAID approach.

1. Introduction

In a RAID 5 disk array, small writes take a long time to complete [Patterson88]. This is known as the “small update problem”. In such an array, redundancy for a stripe of data is provided by a parity block, computed

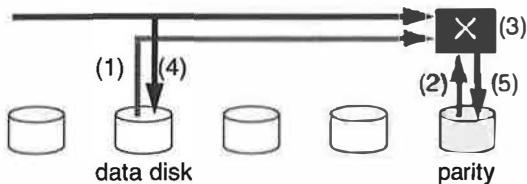


Figure 1: doing a small update in a traditional RAID 5.

as the XOR of the data blocks in the stripe, in order to allow recovery if any disk fails. If a portion of a stripe is updated, the parity data must also be updated to preserve the recoverability property (Figure 1). To do this, it is necessary to (1) read the old value of the data to be overwritten, unless it is already cached in the array controller; (2) read the old parity; (3) XOR the new data with the old, and XOR the result with the old parity to generate the new parity data; (4) write the new data and (5) write the new parity.

Thus, three or four disk I/Os are needed to achieve one small write — all of which are in the critical path. In contemplating this problem we made the following observations:

- modern disks are extremely reliable—so much so that disk array reliability is limited more by its support components than its disks;
- many real workloads have slack periods between bursts of client activity;
- people are already well-used to the notion of time-limited exposure to risk.

These eventually led us to the idea of AFRAID (*A Frequently Redundant Array of Independent Disks*).¹ AFRAID is a RAID 5 disk array that relaxes the coherency between data and parity for short periods of time; parity is made consistent again in the idle periods between bursts of client writes. Thus the stored data is *frequently* held redundantly, rather than always guaranteed to be so.

In this approach, small updates are not required to wait for the parity to be updated, thereby reducing the four I/Os in the critical path of the traditional small-update protocol to just one: write the new data. The benefit is that performance approaches that of an unprotected array. The disadvantage is a slightly increased risk of data loss from a disk failure, but we will show that this increase is small in practice, and also that it can be bounded at the cost of some performance. That is, AFRAID allows a smooth trade-off between increased reliability and increased performance.

¹ Like so many good ideas, ours was of course developed by back-determination from the acronym.

1.1. The AFRAID design

A write in AFRAID does two things: it updates the on-disk copy of the original data, and it causes the target stripes to be marked *unredundant*—i.e., that their parity needs rebuilding. This is indicated by setting a bit per stripe in a non-volatile memory in the array controller; attempting to re-mark an already-marked stripe does nothing. Once one or more stripes have been marked, the AFRAID controller waits until the array is idle and then starts to process the pending parity updates where they can be achieved at effectively zero performance cost to the clients of the array.

Each parity update requires reading all the data blocks in the stripe and XORing them together to generate a new parity block. The new parity block is then written over the top of the old one, after which the unredundant mark for the stripe is removed. The rebuilding of adjacent unprotected stripes can be coalesced to increase the efficiency of disk accesses. Since the overhead of the parity update is linear with the number of disks in a stripe group, AFRAID is best suited to arrays with smaller numbers of disks.

The additional cost to build an AFRAID is just the cost of the marking memory: one bit per stripe. With an array that is 5 disks wide and has a stripe unit size, or stripe depth, of 8 KB, this is ~3 bits per 100 KB, or 3 KB of memory per 1 GB of stored data—a trivial cost compared to the cost of the disk storage itself. The recovery technique for a failed marking memory is simply to rebuild parity for the whole array. This rebuilding can proceed in parallel with continued use.

Any write to a stripe unprotects it all—not just the data being written to. Somewhat counterintuitively, this loss may include data that has not been written to recently. This failure mode is a natural consequence of RAIDs protecting whole stripes, rather than individual blocks. In practice, the exposure is quite small, because the likelihood of data being lost is minimal.

Rather than simply waiting for an idle period before starting to reconstruct parity, it is possible to configure AFRAID to be more aggressive about availability, at the possible cost of greater interference with foreground I/Os. Sample policies include:

- allowing parity rebuilding to start even when there is a non-zero client load on the array;
- giving parity rebuilding priority over foreground client I/Os;
- reverting to normal RAID 5 behavior.

These techniques can be enabled dynamically and adaptively to achieve specified long-term availability and performance goals. We explore the performance and availability effects of some of these policies below.

1.2. AFRAID design assumptions

In this section we provide some additional information about the suppositions on which we based our work:

Disk reliability. Modern disks have published mean time to failure (MTTF) times of 0.5–1 million hours, and this number is increasing every year. As a result, small disk arrays have vanishingly small chances of experiencing a dual-disk failure, which would cause a data loss. The expected disk-related mean time to data loss (MTTDL) in a small RAID 5 with a half-dozen disks is measured in hundreds of millions of hours—several tens of thousands of years. This is far larger than the limits imposed by other “support” components such as the array’s power supply, controller, and cabling. Small disk arrays with less than a dozen disks are the most common in practice, and their overall data availability may not be reduced much if full on-disk redundancy is not provided for short periods.

As we will show, any data-redundancy scheme that produces a disk-related MTTDL of a few million hours or better will be dominated in practice by the array support components. An aggregate MTTDL of a million hours (114 years) translates into only a 2.6% likelihood of any data loss at all during a typical 3-year array lifetime. This is much lower than the rate of problems due to software failures, operator errors, and other environmental difficulties [Gray90, Gray91a]—that is, a small-to-medium sized array that achieves an overall MTTDL of 1M hours or better will probably be entirely adequate for the majority of its applications.

In addition to reduced failure rates, modern disks also provide feedback mechanisms for predicting when such failures will occur. These can warn of impending disk failures hours or days in advance by looking at soft-error logs (e.g., the number of retries required on reads), or the variation in head flying-height. [Lin90c] discusses one such experiment, which was able to predict 93.7% of system failures in a distributed file system, typically many hours before they happened. More recently, IBM disks drives have incorporated a scheme that has been shown to offer a mean of 10 days warning of disk failures [IBMPfa95], with an anticipated success rate of 50–60%. Other manufacturers are following suit. With such techniques available, the likelihood of experiencing an unexpected disk failure can be made very small.

Bursty access patterns. Many (if not most) real uses of disk arrays have bursty access patterns, with periods of relative inactivity between groups of client accesses. [Ruemmler93] offers one quantification of this, in some detail. Indeed, we believe that it is usually only in benchmarks or very large, carefully-tuned systems that arrays are driven to saturation for long periods of time. Given this, there will be spare I/O time available in the idle periods between bursts. If expensive operations such as parity updates can be delayed to

these less busy periods, they can be achieved at little or no apparent performance cost to the client.

Time-limited exposure to data loss. This principle is already well understood and frequently exploited. For example, data in UNIX² file systems is held unprotected in volatile memory buffers before it is written out to disk [Ritchie84a]. Because data is typically only held in volatile memory for short periods of time before being written to disk, this exposure is tolerated in exchange for the increased performance that results. This idea has been extended still further by special file systems that deliberately store data in volatile memory [McKusick90, Ohta90].

Most systems that use non-volatile memory (NVRAM) assume that a single copy of the stored data is sufficient: providing true single-failure-tolerant NVRAM systems is difficult [Menon93], and so is rarely done. Examples of common systems that make this tradeoff include the popular PrestoServe card [Moran90] for NFS servers, as well as recent file system designs, such as LFS [Rosenblum92] and Zebra [Hartman95]. All these rely on assembling large amounts of data in NVRAM to obtain both good performance and acceptable reliability. Other studies have suggested ways to extend this use of NVRAM still further [Baker92b].

Together, these thoughts led us to the AFRAID notion: consciously sacrificing a small amount of data redundancy in order to achieve considerably better performance.

The remainder of the paper explores the AFRAID idea in some detail. We begin with a short discussion of closely-related prior work. This is followed by a description of analytic availability models for AFRAID, and then a quantitative evaluation of the availability and performance effects of the AFRAID design as compared to a traditional RAID 5 and a set of unprotected disks. We conclude with some observations on what this study taught us.

2. Related work

The most obviously related prior solution to the small update problem is parity-logging [Stodolsky93]. A parity-logging array defers the parity-update cost to a later time, at which point it can be performed more efficiently. It does this by performing the traditional RAID 5 read-modify-write operation on the data block being updated, but then, instead of doing the same for the parity block, it writes the XOR of the old and new data to a log—thereby preserving full redundancy all

the time. At a later date, the log file is replayed against the disk array, and the parity updated in situ.

By comparison, AFRAID avoids a pre-read of the old data in the critical path for writes, and thus saves a complete disk revolution on most small writes. It also avoids potentially long delays during parity rebuilds. For efficiency, the parity logging scheme applies a batch of parity updates at a time, which can interfere with foreground I/O requests. Although some of the policies used in AFRAID to control availability can also generate interference with foreground I/Os, they are less intrusive because parity updates may be preempted between stripes.

The parity logging scheme could be extended to apply its parity updates in idle periods, like AFRAID. This would improve its performance, except under workloads in which the parity log fills up, when either the pending parity updates must be applied immediately, interrupting foreground processing to do so, or the array must revert to a regular RAID 5 model of operation until it becomes idle enough to apply updates. Efficiency will drop for either approach. There is no parity log to fill up in AFRAID—all that happens is that the data becomes less well protected.

Finally, parity logging is quite a complicated scheme; AFRAID is much simpler.

Another approach to the same problem is the floating-parity scheme [Menon89]. This reduces the cost of parity updates by writing the new parity data to an empty, rotationally-nearby slot in the array, rather than waiting for a full revolution to go by to update it in place. Such an array still needs to do the old-data and old-parity reads. The floating-data scheme extends this placement optimization to data blocks, too, but this requires considerably larger amounts of non-volatile state information: a word or two per stored block.

In contrast to these two schemes, AFRAID has a less efficient parity update scheme (reading all the data portions of the stripe and recalculating the parity from scratch), but it uses it during a time when the array is less utilized, so that the resulting client-visible cost is small or zero. The result is better performance when the array is active, at the cost of a small exposure to data loss if a (rare) disk failure happens before the new parity has been calculated and written. We quantify the degree of this exposure more precisely in Section 3.

The idea of allowing a file to become unredundant while it was being created was proposed in [Cormen93], in the context of parallel file systems for scientific computing. This paper also suggested the notion of *paritypoints*, by analogy to checkpoints, at which an application could ask for the parity to be computed for the file. Our AFRAID design takes these ideas several steps further: it automates the process of recomputing parity; and does so in idle periods rather than only on demand; it isolates the parity rebuilding

² UNIX is a registered trademark in the United States and other countries, licensed exclusively through X/Open Company Limited.

inside the disk array where it need not be visible to application programs; and it is not limited to stripe-aligned files.

Determining when the array is going to be idle enough to do the rebuild without impacting incoming work is a problem that has been studied already. [Golding95] discusses this problem and a variety of solutions to it.

All the well-known techniques that have been developed for performing stripe rebuilds in a recently repaired disk array can be applied to the problem of rebuilding the parity in AFRAID (e.g., [Muntz90, Holland92]). These include opportunistically piggybacking the parity updates on other “nearby” activity done in the foreground; batching together updates that are physically close together; or simply doing a single, linear sweep through the disks.

Similarly, existing schemes for balancing disk traffic under failure conditions can be applied to AFRAID (e.g., [Gray90c, Muntz90, Blaum94, Reddy91]). For ease of exposition, however, we concentrate here on a straightforward left-symmetric RAID 5 data layout.

3. Availability model of AFRAID

In this section we develop analytic models of data-loss mechanisms for AFRAID, basing them on similar models for traditional RAIDs. In the next section we apply these models to the data from our simulation experiments to provide a quantitative evaluation of data availability in AFRAID.

Following [Gibson93], we do not separate the cases of inaccessible data from data that has been lost irrevocably. We use the term *availability* in this paper to refer to the amount of time that data is accessible and/or not lost. To make our discussion concrete, we apply a set of assumptions about typical failure rates for modern array components; these are summarized in Table 1.

Because manufacturers do not yet publish $MTTF_{\text{unexpected}}$ separately from overall $MTTF$, we have been fairly conservative and set the coverage factor C for disk failure predictions to 0.5 in our calculations. That is, we assume that half of the disk failures will not be predicted ahead of time. In what follows, we include the coverage factor in the $MTTF_{\text{disk}}$ calculations:

$$MTTF_{\text{disk}} = MTTF_{\text{disk-raw}} / (1 - C)$$

3.1. Mean time to first data loss

Most RAID data-loss calculations look only at the time to the first catastrophic data loss due to disk failure, which occurs if two disk failures occur so close together that the first failure has not yet been recovered from. If this happens, two disks worth of data is lost. A convenient measure for this kind of catastrophe is the *mean time to data loss* (MTTDL). The equation for a

Table 1: values assumed for calculations in this paper.

Parameter	Value
disk mean time to failure $MTTF_{\text{disk-raw}}$	1 M hours
support hardware mean time to data loss $MTTDL_{\text{support}}$	2M hours
disk failure-prediction coverage (C)	0.5
mean time to repair (MTTR)	48 hours
stripe unit size (S)	8KB
size of disk (V_{disk})	2GB

RAID 5 disk array with $N+1$ disks, assuming rare, independent, exponentially-distributed disk failures is:³

$$MTTDL_{\text{RAID-catastrophic}} = (MTTF_{\text{disk}})^2 / (N(N+1) \times MTTR_{\text{disk}}) \quad (1)$$

With a 5-disk array, and the parameters of Table 1, this gives a theoretical MTTDL of $\sim 4.10^9$ hours, or about 475,000 years.

In addition to the regular RAID failure mode, AFRAID exhibits data loss if a single disk fails unexpectedly while there is some unprotected data. To determine the combined effect of these two modes, we look at the likelihood of data loss occurring when there is unprotected data (T_{unprot}) and when there is not ($T_{\text{total}} - T_{\text{unprot}}$). A conservative measure of the contribution to MTTDL for the period in which there is unprotected data is:

$$MTTDL_{\text{AFRAID-unprotected}} = (T_{\text{total}}/T_{\text{unprot}}) \times MTTF_{\text{disk}}/(N+1) \quad (2a)$$

This measure is conservative because we do not take account of the fact that in some cases only parity data will be lost: we just simplify and assume that there will always be some data loss. The rest of the time, when there is no unprotected data, AFRAID behaves just like a RAID for the MTTDL measure:

$$MTTDL_{\text{AFRAID-RAID-catastrophic}} = T_{\text{total}}/(T_{\text{total}} - T_{\text{unprot}}) \times MTTDL_{\text{RAID-catastrophic}} \quad (2b)$$

Summing these two contributions, which are best thought of as inverses of rates, gives:

$$MTTDL_{\text{AFRAID}} = 1 / (1/MTTDL_{\text{AFRAID-unprotected}} + 1/MTTDL_{\text{AFRAID-RAID-catastrophic}}) \quad (2c)$$

³ [Gibson93] includes several rather fancier formulae (e.g., equations 12 and 14) that give additional accuracy for large arrays with many tens to hundreds of disks. In addition to the fact that it would need another page or so to explain them, they don't help characterize the much smaller arrays that are the common case, and the target of AFRAID.

Section 4.3 presents the results of experimental determinations of this value over several workloads.

One more kind of multiple failure can afflict an AFRAID: if its NVRAM marking memory fails, the array will start reconstructing parity across all the stripes. This will take a little while (about ten minutes for an array using 2GB disks that can read at a sustained rate of 5MB/s). If a disk failure occurs before the parity has been completely rebuilt, the array has no way of knowing which stripes were still unprotected, if any, although it will be bounded by the knowledge of how far the reconstruction has progressed. The likelihood of this failure is exceedingly small, however, because of the small window of vulnerability ($\text{MTTDL} > 10^{11}$ hours), so we can safely ignore it here.⁴

3.2. Mean data loss rate

The MTTDL measure indicates the expected rate of failures leading to *any* data loss. In any RAID 5-based system this occurs on a dual-disk failure, at which point a catastrophe occurs: two whole disks worth of data vanishes. In addition, unprotected data in AFRAID is vulnerable to loss from a single-disk failure. However, the amount of data lost in this case is bounded by how much is unprotected—and we will show later that it is often quite small.

There is an important qualitative difference between losing a block or two on a disk and losing the whole disk. For example, all disks have a few defective sectors or tracks, and new ones are occasionally added to this list over its lifetime—but the occurrence of a bad block on a disk doesn't mean that the entire disk has been lost, or even that it should be discarded. Similarly, the effect of accidentally deleting a single small file is usually much less severe than that of losing an entire file system. The former may be merely tedious, while the latter can be a catastrophe.

There are several reasons for this qualitative difference: not all data is equally valuable; some data can easily be reconstructed or recomputed; much data “dies young”—that is, it will be deleted or overwritten soon after it is created [Ousterhout85a]; recovering a single file is often simpler than rebuilding an entire disk set. Others have taken advantage of this difference before us. For example, the BSD fast file system [McKusick84] and its journaling file system successors take considerable care to maintain consistency of file system metadata, but are much more cavalier with user information.

As a result, we feel that it is important to measure the *amount* of data subject to loss, as well the time to lose the first byte. A good metric for this is the *mean data loss rate* (MDLR): the product of the amount of data

⁴ The formula, for the curious, is:

$$\begin{aligned}\text{MTTDL}_{\text{NVRAM+disk}} = & \text{MTTF}_{\text{NVRAM}} \\ & \times \text{MTTF}_{\text{disk}} / ((N+1) \times \text{rebuild-time})\end{aligned}$$

loss and the rate at which it is likely to occur. In addition to quantifying the effects of such losses, it has the advantage of being a reminder that mean time to failure values should be used only to define failure rates, not expectations of *lifetimes*.

The catastrophic data loss rate for a regular array due to a two-disk failure can be cast in these terms as:

$$\begin{aligned}\text{MDLR}_{\text{RAID-catastrophic}} = & \\ 2V_{\text{disk}} \times N/(N+1) \times 1/\text{MTTDL}_{\text{RAID-catastrophic}} & (3)\end{aligned}$$

where V_{disk} is the capacity of a single disk, which is reduced by the second term to reflect that some of the lost disk space held parity rather than data blocks. The RAID 5 array we considered earlier would have a MDLR of ~0.8 bytes/hour from this failure mode.

Analyzing the impact of single disk failures in AFRAID requires additional information. To provide a basis for this availability analysis, we first introduce the notion of *parity lag*, which is the amount of unredudant non-parity data present in the array at any time, measured in bytes. The *mean parity lag* is the average parity lag over some test period, such as the duration of a test workload. Note that parity lag is workload dependent.

Data loss only occurs from a single disk failure if the parity lag is non-zero at the time of the failure. When this happens, one stripe unit (block) from each unredudant stripe is lost (the one on the failed disk), unless the lost block is a parity block, in which case no actual data loss occurs.

The mean data loss rate for a single-disk failure on AFRAID with unprotected data is:

$$\begin{aligned}\text{MDLR}_{\text{unprotected}} = & \\ (\text{mean-parity-lag}/N) \times (N+1)/\text{MTTF}_{\text{disk}} & (4)\end{aligned}$$

where the first term reflects the average amount of unprotected non-parity data vulnerable to a single disk failure, and the second term gives the total failure rate of all the disks in the array. We will present experimental determinations of these values in section 4.3.

Summing the different component contributions gives us the final mean data loss rate for the disk-related components of AFRAID:

$$\begin{aligned}\text{MDLR}_{\text{AFRAID}} = & \\ \text{MDLR}_{\text{RAID-catastrophic}} + \text{MDLR}_{\text{unprotected}} & (5)\end{aligned}$$

3.3. The effect of support components

We have concentrated so far only on disk failures. This emphasis made sense when disks were much less reliable than support components such as cooling fans, power supplies, cabling, and other passive components. But one of our contentions is that disks are no longer the primary cause of problems in a disk array. The reliability of the support hardware and the

array controller is little or no better than that of the disks.

The data in [Schulze89] suggests that these support components would together lead to a mean time to failure of a small array of about 46k hours; [Gibson93] simply increases this to “a more reasonable value of 150k hours” without further discussion. Fortunately, more recent designs and pressure by manufacturers to boost reliability seem to have increased the quality of these components, and although many array manufacturers disconcertingly consider the data either irrelevant or proprietary, a few do not, and we were quoted MTTF numbers of 20–35k hours, and MTTDL values of 270k to 5M hours. Some typical component MTTF examples are: 150k or 0.5–1M hours for the controller; 300–500k hours for a host bus adapter; 50–350k hours for a power supply module; and 1–3M hours for cabling and packaging.⁵

It takes considerable engineering effort and use of redundant components to increase the overall MTTDL above 1M hours. For example, the HP Autoraid array [Wilkes95] uses two redundant power supplies, three fans (any two of which can keep the system cool enough for continued operation), and can support a dual controller configuration; each controller has a separate NVRAM that uses dual rechargeable batteries that are periodically discharge-tested. The result: with a fully-populated system (12 disks and 2 controllers), the array’s overall MTTDL is specified (probably conservatively) as 1.97M hours, together with an overall MTTF of 31k hours. Few designers of small arrays go to all this trouble: for example, the Network Appliance’s FA Server350 product has a predicted MTTF of around 20–30k hours with four disks, and disks are its only redundant components.⁶

Together, these figures suggest that the current “more reasonable value” for the aggregated non-disk components of a *conservatively-engineered* array is probably about 2M hours. This is a far cry from the 4.10^9 hours calculated from the independent-disk failure model considered earlier. With a 2M hour MTTDL, our 5-disk array would suffer a MDLR of 4.0KB/hour; using the 150k hour figure from [Gibson93] would increase this to 53KB/hour.

The lesson here is that it is the support components that determine the availability of a modern disk array, not its disks.

3.4. Non-volatile memory

Despite the extensive use of NVRAM in high-availability systems, remarkably little data has been published on its reliability.

⁵ Storage Dimensions technical support line, personal communication, October 1995.

⁶ Rich Boburg, Network Appliance, personal communication, October 1995.

Integral lithium-cell-backed static RAM is probably one of the most reliable kinds of NVRAM: it offers retention lifetimes of 25–87k hours and extremely low failure rates [Dallas94, Dallas95], but it is quite expensive: ~\$350/MB for a representative state-of-the-art part from Dallas Semiconductor.

To avoid this expense, many systems use dynamic RAM backed by rechargeable batteries based on NiCd cells. The battery technology often limits the resulting availability: achieving MTTF values above a few tens of thousands of hours requires the use of redundant batteries whose status is periodically tested by controlled discharging, and careful attention to charging circuitry and battery conditioning. The complexity and cost of this design means that it is not often used, so most battery-backed NVRAM has a much lower MTTF than the Li-cell backed RAM. For example, the popular PrestoServe card has a predicted MTTF of 15k hours [Neary91]; with 1MB of vulnerable data, this corresponds to an MDLR of 67 bytes/hour.

We will show that this means that single-copy NVRAM applications are already accepting significantly higher risk of data loss than results from the temporary lack of parity protection in AFRAID.

3.5. Power failure

One additional support component that is particularly important is external power. Up to this point, our discussion has assumed that external power failures simply don’t happen. This matters because a power failure that happens while a RAID 5 is writing can lead to data loss unless a separate, non-volatile intentions log is kept.

[Gibson93] reports a MTTF of 4300 hours for mains power (i.e. a power failure about every 6 months). This is probably reasonable for parts of North America and Europe, but would be optimistic in some other parts of the world. In our traces, we saw outstanding writes up to 59% of the time, with a mean of 20%. Even using a more conservative value of a 10% write duty cycle on a 5-disk RAID 5 gives a MTTDL of only 43k hours due to external power failures. The effect on MDLR is roughly to double it (0.7bytes/hour), but the change in MTTDL represents losing about 98% of the availability that the array offers.

It might be thought that providing an uninterruptible power supply, or UPS, would be overkill for a small array, but it may be the single largest contributor to preventing data loss. Using a high-grade UPS with an MTTF of 200k hours [Best95] and a 10% write duty cycle returns the MTTDL for the array’s external power components to 2M hours.

The large variability in power and UPS reliability can obscure the other support contributions to MTTDL, so we have chosen not to include external power failure in the calculations in this paper.

3.6. How much availability is enough?

When RAIDs were first being discussed as a replacement for large disks, the MTTF for small disks was 20–30,000 hours, and the target was to match the reliability of a single, large disk with a MTTF of 30–100,000 hours [Patterson88]. Things have improved since then: modern small-form-factor disks typically have a published MTTF of 0.5–1.0M hours. Given that the expected useful lifetime of a disk or disk array is probably no more than 3 years, or about 26k hours, this is equivalent to a lifetime expected failure likelihood of 3–5%. If it held 2GB, its mean data loss rate would be 2–4KB/hour. This means that the best of the traditional 5-disk RAIDs, limited to a MTTDL of about 1–2M hours by their support components, are achieving a MDLR for the whole array roughly equivalent to that of a single disk.

We contend that the combination of modern, highly reliable disks with traditional RAID technology provides more than enough protection against disk failures, and that further efforts to increase data availability are attacking a solved problem for the vast majority of customers. Instead, we suggest that it may be worth exchanging some of the “excess” disk availability for better performance—which is precisely what AFRAID does.

4. Evaluation

To provide a quantitative evaluation of the AFRAID concept, we used a detailed event-driven simulator to compare the performance and availability of an AFRAID array with a non-AFRAID system under a variety of workloads. We report here on three aspects of this evaluation:

- the relative performance of AFRAID, RAID 5, and RAID 0 (an unprotected array);
- quantitative availability measures;
- the relationship between performance and availability under AFRAID.

We begin with a description of our experimental setup.

4.1. Experimental methodology

In order to evaluate whether real-life workloads are bursty enough for an AFRAID array to rebuild parity quickly, it was necessary to look at some real-life workloads. So we did. Here are the ones we used:

- *hplajw* — a single user HP-UX [Clegg86] system used mainly for email and document editing.
- *snake* — an HP-UX file server for a cluster of workstations at UC Berkeley.
- *cello* — an HP-UX timesharing system for about 20 people doing text editing and program development. We used two subsets of the full cello trace: *cello-usr* is the set of three disks holding the

root file system, /usr, and /users; *cello-news* is a single disk holding the Usenet news database: it received half of all the disk I/Os in the system.

- *netware* — an intensive database-loading benchmark measured on a Novell Netware server.
- *ATT* — a production telephone-company database system. On the real system, the entire dataset was mirrored; for our tests, we just used one copy of the data.
- *IBM AS400* — four production AS400 systems. These traces were supplied to us by Bruce McNutt of IBM San Jose; we called them AS400-1 through AS400-4.

The workloads for the first three of these systems are described in great detail in [Ruemmler93]. We used one-day subsets of them for this work.

To evaluate our claims we constructed a detailed event-driven performance simulation of AFRAID using the Pantheon⁷ simulator, which includes the calibrated disk models discussed in [Ruemmler94]. To simplify the discussions and save space, we just consider spin-synchronized arrays here.

To add AFRAID to Pantheon, we started with a detailed RAID 5 model and adapted it to support AFRAID. The changes were small: they consisted of adding the marking memory and updating it on writes, and not doing the read-modify-write cycle for the parity in AFRAID mode. We added a background idle-task to do the parity rebuilds, triggered by an idle-detection network [Golding95] or explicit foreground policies. By default, we used a timer-based idleness detector with a 100ms delay: that is, AFRAID started processing parity updates once the array had been completely idle for 100ms; the output from the idle-period predictor was ignored.

To make sure that we were seeing the effects of the AFRAID policies themselves rather than just the disk array’s cache policies [Ruemmler94], we chose a small (256KB) write staging area with a write-through policy together with a small (256KB) read cache with no array-level readahead. Since our workloads came from systems with much larger file buffer caches, read hits in the array’s cache were rare. We limited the number of concurrently active client requests inside the array to the number of physical disks it had; the host device driver used the CLOOK policy [Worthington94a], the back-end device drivers inside the array used FCFS. We modelled HP C3325 2GB 3.5” 5400 RPM disks in the array [HPC3324].

Multiple writes to the same stripe were allowed to proceed in parallel, but would block if a parity-rebuild

⁷ The simulator used to be known as TickerTAIP: we changed its name to avoid confusion with the parallel RAID array of the same name [Cao94b].

on that stripe was in progress. Requests were never preempted: once started, they ran to completion.

The I/O times we report in this paper start when a request is given to the device driver, and stop when the request is completed by the array. They include both the time spent in the array itself and any time spent queued in the device driver. Given that we are using an open-queueing, trace-driven workload, this provides the fairest assessment of the performance that would be seen by a user or file system.

We took no special action for synchronous writes: ones for which the file system waits until the data being written has been put onto non-volatile media. Such writes are designed to ensure resilience against power-failure, not against disk failure; for example, they are used to disable immediate-reporting in disks that allow this [Ruemmler93, Ruemmler94]. Even if we had chosen to force a parity-update on a stripe updated by a synchronous write, the redundancy would go away again on the next update to any block in the stripe—not just the one that had been written to synchronously—because parity protection is at the stripe level, not the block level.

Because almost all of the code was the same between the various array models, direct performance comparisons between them are possible. Indeed, to make sure that the exact same disk and cache algorithms was executed in all cases, we modelled RAID 0 as an AFRAID that simply never did parity updates.

About the only things that we did not model were a few performance improvements for AFRAID, of which the most important were probably aggregation of adjacent stripes needing parity rebuilds and piggybacking parity updates on disk accesses to nearby blocks.

In addition to the baseline AFRAID design, which updated parity only in idle periods, we implemented a

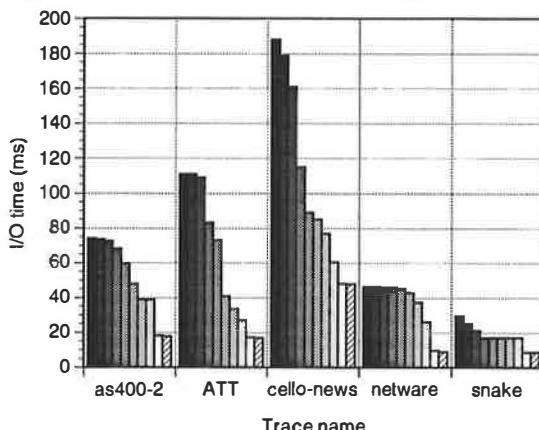


Figure 2: absolute performance of RAID 5 (leftmost bars), AFRAID-baseline and RAID 0 (rightmost bars) with a range of AFRAID-MTTDL-X policies in between.

policy which directly traded off performance for improved availability. This MTTDL-X policy is designed to keep the disk-based MTTDL above a particular target value (x). To do this, it continuously calculates the MTTDL that has been achieved so far, and reverts to RAID 5 mode if the goal is not being met. (It also starts the parity update for any unprotected stripes at this time.) The policy attempts to limit MDLR by automatically starting a parity update when more than 20 stripes are unprotected, even if the array is not idle; we had found earlier that this was fairly effective and caused little performance degradation.

4.2. Performance evaluation of AFRAID

Figure 2 and Table 2 present the results of exploring the relative performance of AFRAID, RAID 5 and RAID 0 across a range of workloads and parity-update policies. The figure and table show that, as predicted, pure AFRAID performance is very close to that of RAID 0, with a smooth degradation in performance towards that of RAID 5 as AFRAID is configured to increase data availability.

The performance of the baseline AFRAID was a geometric mean of 4.1 times that of RAID 5 across our test workloads. By comparison, RAID 0 performance was 4.2 times that of RAID 5. Thus, AFRAID is living up to the first part of its promise: performance comparable to non-protected arrays.

4.3. Availability measures for AFRAID

Our next experiments determined the availability delivered by the different parity update policies under real workloads. The results are shown in Table 3 and Table 4. The AFRAID contribution to MDLR from unprotected data is extremely low: with the exception of the heavy load from the ATT trace, $MDLR_{unprotected}$ contributes less than one byte per hour to the overall MDLR. This is tiny by comparison to the overall MDLR,

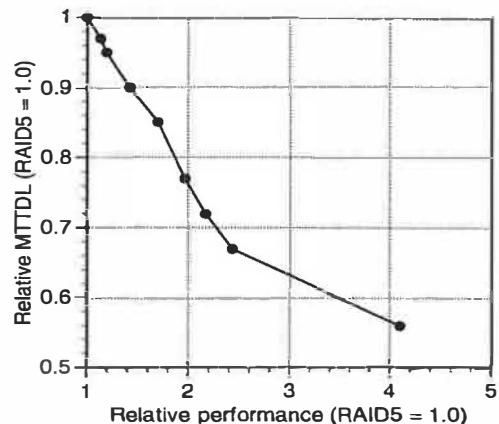


Figure 3: relative performance and MTTDL for RAID 5 (top left) to AFRAID-baseline (bottom right). These are geometric means across all the workloads we studied.

Table 2: performance data for the traces we studied.
AFRAID-MTTDL-X is an AFRAID that reverts to RAID 5 when the availability drops below a target threshold.

Workload	AS400-1	AS400-2	AS400-3	AS400-4	ATT	cello-news	cello-usr	hplajw	netware	snake
Number of disks (N+1)	4	4	4	4	5	4	4	4	8	4
Trace duration (hours)	0.6	1.5	1.7	1.0	1.0	24	24	24	1.1	24
Mean I/O time (milliseconds)										
RAID0	58.0	18.0	12.8	22.6	17.1	48.1	13.0	20.9	8.9	8.7
AFRAID-baseline	58.5	18.4	12.9	22.9	17.5	48.2	13.2	21.2	9.7	8.8
AFRAID-MTTDL-2M	125	38.9	23.8	39.3	33.6	77.1	18.5	27.2	37.4	17.2
AFRAID-MTTDL-16M	179	68.2	33.5	73.6	83.2	115	43.3	27.2	45.9	17.2
AFRAID-MTTDL-64M	183	73.8	37.2	79.9	111	179	96.8	27.2	46.4	25.1
RAID5	183	74.2	37.8	80.4	111	188	104	70.2	46.4	29.7

Table 3: mean data loss rate (MDLR) for the traces we studied.
MDLR-nosupport excludes data losses due to the support hardware, while MDLR-total includes them.

Workload	AS400-1	AS400-2	AS400-3	AS400-4	ATT	cello-news	cello-usr	hplajw	netware	snake
<i>MDLR-unprotected (bytes/hour)</i>										
AFRAID-baseline	0.08	0.02	0.01	0.02	5.93	0.47	0.06	<0.01	0.71	0.02
<i>MDLR-nosupport (bytes/hour)</i>										
RAID0	18K	18K	18K	18K	32K	18K	18K	18K	98K	18K
AFRAID-baseline	0.51	0.45	0.44	0.46	6.70	0.90	0.50	0.43	3.06	0.45
RAID5	0.43	0.43	0.43	0.43	0.77	0.43	0.43	0.43	2.35	0.43
<i>MDLR-total (bytes/hour)</i>										
RAID0	21K	21K	21K	21K	36K	21K	21K	21K	105K	21K
RAID5, AFRAID	3K	3K	3K	3K	4K	3K	3K	3K	7K	3K

Table 4: mean time to data loss (MTTDL) data for the traces we studied.
MDLR-nosupport excludes data losses due to the support hardware, while MDLR-total includes them.

Workload	AS400-1	AS400-2	AS400-3	AS400-4	ATT	cello-news	cello-usr	hplajw	netware	snake
<i>Percentage of time with unprotected data</i>										
AFRAID-baseline	51.1%	18.3%	13.4%	22.4%	22.5%	7.9%	8.8%	<0.1%	51.3%	4.2%
AFRAID-MTTDL2M	25.6%	12.7%	7.4%	13.7%	10.7%	3.9%	3.2%	<0.1%	12.3%	2.1%
AFRAID-MTTDL64M	0.8%	0.8%	0.8%	0.8%	0.6%	0.7%	0.7%	<0.1%	0.3%	0.7%
<i>MTTDL-nosupport (hours)</i>										
RAID0	0.33M	0.33M	0.33M	0.33M	0.25M	0.33M	0.33M	0.33M	0.14M	0.33M
AFRAID-baseline	0.98M	2.74M	3.74M	2.23M	1.77M	6.29M	5.63M	300M	0.49M	11.8M
AFRAID-MTTDL2M	1.95M	3.92M	6.74M	3.66M	3.74M	12.8M	15.5M	556M	2.02M	24.0M
AFRAID-MTTDL64M	61.2M	63.5M	63.6M	62.9M	62.9M	66.4M	66.4M	556M	64.6M	75.3M
RAID5	6.94G	6.94G	6.94G	6.94G	4.17G	6.94G	6.94G	6.94G	1.49G	4.17G
<i>MTTDL-total (hours)</i>										
RAID0	0.29M	0.29M	0.29M	0.29M	0.22M	0.29M	0.29M	0.29M	0.13M	0.29M
AFRAID-baseline	0.66M	1.16M	1.30M	1.05M	0.94M	1.52M	1.48M	1.99M	0.39M	1.71M
AFRAID-MTTDL2M	0.99M	1.32M	1.54M	1.29M	1.30M	1.73M	1.77M	1.99M	1.01M	1.85M
AFRAID-MTTDL64M	1.94M	1.94M	1.94M	1.94M	1.94M	1.94M	1.94M	1.99M	1.94M	1.95M
RAID5	2.00M	2.00M	2.00M	2.00M	2.00M	2.00M	2.00M	2.00M	2.00M	2.00M

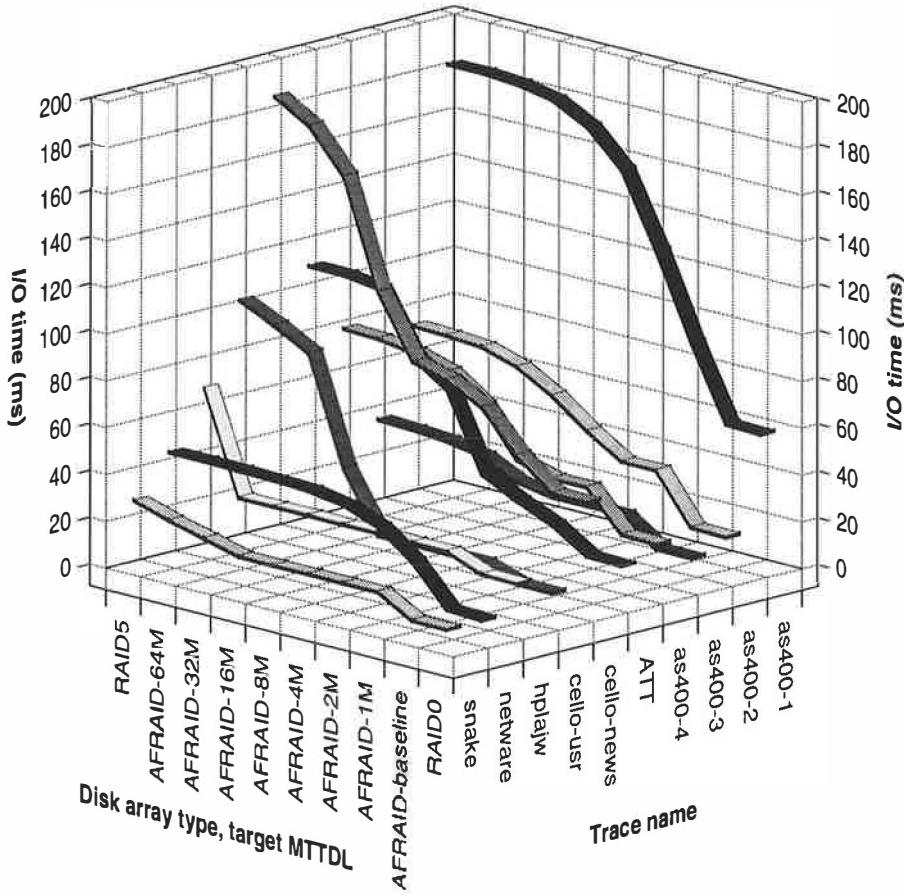


Figure 4: performance of RAID 0, RAID 5 and AFRAID under different workloads and policies.

which is dominated by support-component effects. Consequently, AFRAID and RAID 5 have essentially identical MDLRs. The MDLR_{unprotected} drops to less than 0.1 bytes/hour if any of the MTTDL-X policies are used.

Because the MDLR values for AFRAID are so close to those for RAID 5, the more interesting comparison is between the MTTDL values. The first thing to note is that even the baseline AFRAID design is uniformly better than an unprotected disk array. It delivers a geometric mean MTTDL 4.3 times better than RAID 0, and is only a factor of 1.8 worse than pure RAID 5.

The MTTDL-X policy can bring the overall AFRAID MTTDL as close to RAID 5 as desired. Even the simple implementation of this policy that we used proved highly effective: the disk-related MTTDL was never more than 5% below its target, and usually far exceeded it.

As with MDLR, the dominant factor in overall MTTDL comes from the support components, which limit overall MTTDL to 2 million hours for all but the baseline AFRAID with the busiest workloads.

Thus, AFRAID is living up to the second part of its promise: availability comparable to RAID 5.

4.4. How changing availability affects performance

Figure 3 indicates just how little of the availability of a RAID 5 is relinquished by AFRAID in order to obtain better performance. The graph indicates relative performance and availability (MTTDL) by comparison to RAID 5 (the top left data point); it uses the geometric mean of the results obtained from all of our workloads. As the target MTTDL-X value is reduced (points further to the right), performance increases rapidly, while availability drops off much more slowly. For example, AFRAID offers 42% better performance for only 10% less availability, and 97% better for 23% less. By the time pure AFRAID is reached at the bottom right of the graph, performance is 4.1 times better than RAID 5, at a cost of less than half its availability.

Thus, a great deal of performance improvement can be had for a small reduction in data availability.

Figure 4 shows how performance varies with the parity-update policy for each of the traces that we studied. This figure highlights what AFRAID is all about: providing a choice between more performance or more availability.

The tradeoff between performance and availability is directly related to the characteristics of the workload. For instance, the highly bursty workloads such as snake, hplajw, and cello-usr show relatively little change in mean I/O time as availability is increased by choice of more conservative MTTDL-X policies. This is because the workloads have enough idle time to update unredudant stripes and therefore the amount of unprotected time usually stays low; in turn, this means that there is little need to revert to RAID 5 mode. In workloads with fewer idle periods and more write traffic, such as AS400-1 and ATT, there is a smooth decline in mean I/O time as MTTDL is increased across the entire range between RAID 5 and pure AFRAID.

This adaptability is one of the key features of AFRAID. Once a desired level of availability has been specified, an AFRAID array will translate any unneeded redundancy into performance. A typical bursty workload will show performance close to that of an unprotected RAID 0 disk array, while even the most highly utilized workload will deliver performance no worse than a RAID 5.

The net result is that AFRAID lives up to the last part of its promise: it offers a smooth trade-off between performance and protection that a regular RAID cannot.

5. Refinements of the AFRAID ideas

This section suggests some further applications and refinements of the AFRAID idea.

An array could begin in a “conservative” RAID 5 mode, and automatically switch into AFRAID behavior once it had determined that the I/O patterns included sufficient idle time to keep the redundancy deficit below some bound. This would be a more conservative scheme than the one we used in the MTTDL-X policy, which took the opposite approach, switching into RAID 5 when it felt that its target could not be achieved.

Stripe-aligned subsets of an AFRAID’s storage space could be permanently flagged with different redundancy properties, from full RAID 5 redundancy-preservation to zero-redundancy RAID 0-style storage. Data could then be mapped to portions of the array that provided different redundancy guarantees, allowing fine-tuning of the array’s availability properties according to user-specified goals [Wilkes91]. The host could then actively request that a set of stripes be made redundant, analogous to the traditional database COMMIT operation.

The units of parity-reconstruction can have a smaller “height” than the stripes used for data layout if more marker memory can be provided. For example, if M memory bits can be afforded per stripe, then parity computations will still be efficient for small writes that update only $1/M$ of a stripe unit.

A RAID 6 array keeps two parity blocks for each stripe, and thus pays an even higher penalty for doing small

updates than does RAID 5. The AFRAID technique could be combined with the RAID 6 parity scheme to delay either or both parity-block updates: if only one was deferred, partial redundancy protection would be available immediately, and full redundancy once the parity-rebuild happened for the other parity block.

6. Conclusions

The main AFRAID idea is the notion of allowing deliberate, controlled, temporary non-redundancy in a disk array in order to get significantly better performance. Because real-life workloads are very often bursty, these performance gains can be achieved with a minimally increased chance of data loss—and indeed, there may be less exposure to data loss than existing single-point-of-failure solutions such as single-copy volatile or NVRAM caches. AFRAID also offers a choice that has not been possible before: that of selecting just how much availability is wanted in a particular situation.

The AFRAID design appears to be highly appropriate for workloads that have even moderate amounts of idle time between bursts of activity. Like other RAID designs, there are some workloads and applications for which it is not particularly well suited. For example, we would not advocate AFRAID for the cases where data must be protected at all costs, but it does offer a very good solution for the majority of people who want something between completely unprotected data and a fully-redundant, high-end disk array with its performance, purchase, and configuration costs.

In particular, we believe AFRAID is an appropriate design for low-load environments where latency is important, such as systems with a small number of interactive users. We hypothesize that these applications are also the ones least likely to benefit from the full availability improvements of RAID 5.

What did we learn as a result of this study? In addition to the performance and availability results we have described already, a few lessons stand out:

- Throughout this paper we have been attempting to reinforce a larger point that deserves more attention in system design: there is little value in bolstering the fault-tolerance of a single component to heroic levels if the rest of the system is less reliable. We call this the *end-to-end availability argument*, by analogy with [Saltzer81]. Making simplifying assumptions about end-to-end availability (for example, that complete data redundancy in the disk layer of an array is sacrosanct, or that NVRAM storage never fails) prevents taking advantage of performance opportunities like AFRAID.
- Real-life workloads really are bursty (we’ve been saying this for a while, but it bears repeating).

- Although the amount of unprotected data in the array is a function of the workload, there are several algorithms for bounding it, at the cost of some of the performance gains from pure AFRAID. Unbounded AFRAID and pure RAID 5 are simply different points on a continuum of allowed parity lag—and our design allows a user to choose where on this scale they would like their array to be.
- Thinking of different availability solutions in terms of data-loss-rate proved a useful way to unify a number of effects.

Finally, just because an idea has a strange acronym doesn't mean you should be worried by it:

"Always do what you are afraid to do."

- Ralph Waldo Emerson

Acknowledgments

Richard Golding, Chris Ruemmler, Carl Staelin, Tim Sullivan, and Bruce Worthington built much of the experimental infrastructure on which this work was based. Denny Georg funded our pursuit of an oddball idea with commendable(?) alacrity. Terri Watson provided the transportation to Orcas Island's Mount Constitution in which the idea took shape, and gave us invaluable feedback on drafts of this paper. Bruce McNutt provided us with the IBM AS400 traces. Our USENIX paper shepherd, Miche Baker-Harvey, also provided many useful comments.

References

- [Baker92b] Mary Baker, Satoshi Asami, Etienne Deprit, John Ousterhout, and Margo Seltzer. Non-volatile memory for fast, reliable file systems. *Proceedings of 5th International Conference on Architectural Support for Programming Languages and Operating Systems* (Boston, MA, 12–15 October 1992). Published as *Computer Architecture News*, 20(special issue):10–22, October 1992.
- [Best95] Best Power Technology Incorporated, Necedah, WI. *Product catalog*, September 1995.
- [Blaum94] Mario Blaum, Jim Brady, Jehoshua Bruck, and Jai Menon. EVENODD: an optimal scheme for tolerating double disk failures in RAID Architectures. *Proceedings of 21st International Symposium on Computer Architecture* (Chicago, IL). Published as *Computer Architecture News*, 22(2):245–54, 18–21 April 1994.
- [Clegg86] Frederick W. Clegg, Gary Shiu-Fan Ho, Steven R. Kusmer, and John R. Sontag. The HP-UX operating system on HP Precision Architecture computers. *Hewlett-Packard Journal*, 37(12):4–22, December 1986.
- [Cormen93] Thomas H. Cormen and David Kotz. Integrating theory and practice in parallel file systems. *Proceedings of the 1993 DAGS/PC Symposium*, (Hanover, NH), pages 64–74, Dartmouth Institute for Advanced Graduate Studies, June 1993.
- [Dallas94] Dallas Semiconductor, Dallas, TX. *Using nonvolatile static RAMs*, Application note 63, September 1994.
- [Dallas95] Dallas Semiconductor, Dallas, TX. *DS1250Y/AB 4096K nonvolatile SRAM: data sheet*, October 1995.
- [Gibson93] Garth A. Gibson and David A. Patterson. Designing disk arrays for high data reliability. *Journal Parallel and Distributed Computing*, 17(1–2):4–27. Academic Press, Incorporated, January/February 1993.
- [Golding95] Richard Golding, Peter Bosch, Carl Staelin, Tim Sullivan, and John Wilkes. Idleness is not sloth. *Proceedings of Winter USENIX 1995 Technical Conference* (New Orleans, LA), pages 201–12. Usenix Association, Berkeley, CA, 16–20 January 1995.
- [Gray90] Jim Gray. *A census of Tandem system availability between 1985 and 1990*. Technical Report 90.1. Tandem Computers Incorporated, September 1990.
- [Gray90c] Jim Gray, Bob Horst, and Mark Walker. Parity striping of disc arrays: low-cost reliable storage with acceptable throughput. *Proceedings of 16th International Conference on Very Large Data Bases* (Brisbane, Australia), pages 148–59, 13–16 August 1990.
- [Gray91a] Jim Gray and Daniel P. Siewiorek. High-availability computer systems. *IEEE Computer*, 24(9):39–48, September 1991.
- [Hartman95] J. H. Hartman and J. K. Ousterhout. The Zebra striped network file system. *ACM Transactions on Computer Systems*, 13(3):274–310, August 1995.
- [Holland92] Mark Holland and Garth A. Gibson. Parity declustering for continuous operation in redundant disk arrays. *Proceedings of 5th International Conference on Architectural Support for Programming Languages and Operating Systems* (Boston, MA). Published as *Computer Architecture News*, 20(special issue):23–35, 12–15 October 1992.
- [HPC3324] Hewlett-Packard Company, Boise, Idaho. *HP C3324/C3724, HP C3325/C3725 3.5-inch SCSI-2 disk drives: technical reference manual*, Part number 5963-0277, edition 3, February 1995.
- [IBMpfa95] Predictive Failure Analysis: advanced condition monitoring. International Business Machines Corporation, World-wide web page <http://www.almaden.ibm.com/storage/oem/tech/predfail.htm>, 21 August 1995.
- [Lin90c] T.-T. Y. Lin and D. P. Siewiorek. Error log analysis: statistical modeling and heuristic trend analysis. *IEEE Transactions on Reliability*, 39(4):419–32, October 1990.
- [McKusick84] Marshall K. McKusick, William N. Joy, Samuel J. Leffler, and Robert S. Fabry. A fast file system for UNIX. *ACM Transactions on Computer Systems*, 2(3):181–97, August 1984.
- [McKusick90] Marshall Kirk McKusick, Michael J. Karels, and Keith Bostic. A pageable memory based filesystem. *UKUUG Summer 1990* (London), pages 109–15. United Kingdom UNIX systems User Group, Buntingford, Herts, 9–13 July 1990.

- [Menon89] Jai Menon and Jim Kasson. *Methods for improved update performance of disk arrays*. Technical report, RJ 6928 (66034). IBM Almaden Research Center, San Jose, CA, 13 July 1989. Declassified 21 Nov. 1990.
- [Menon93] Jai Menon and Jim Courtney. The architecture of a fault-tolerant cached RAID controller. *Proceedings of 20th International Symposium on Computer Architecture* (San Diego, CA), pages 76–86, 16–19 May 1993.
- [Moran90] J. Moran, R. Sandberg, D. Coleman, J. Kepcs, and B. Lyon. Breaking through the NFS performance barrier. *Proceedings of EUUG Spring 1990* (Munich, Germany), pages 199–206, 23–27 April 1990.
- [Muntz90] Richard R. Muntz and John C. S. Lui. Performance analysis of disk arrays under failure. *Proceedings of 16th International Conference on Very Large Data Bases* (Brisbane, Australia), pages 162–73, 13–16 August 1990.
- [Neary91] Annette M. Neary. MM-1350 reliability prediction. Micro Memory Incorporated, Chatsworth, CA, 15th March 1991. Personal communication from Dennis Doe, 23rd Oct. 1995.
- [Ohta90] Masataka Ohta and Hiroshi Tezuka. A fast /tmp file system by delay mount option. *1990 Summer USENIX Technical Conference* (Anaheim, California, June 1990), pages 145–50. USENIX, June 1990.
- [Ousterhout85a] John K. Ousterhout, Hervé Da Costa, David Harrison, John A. Kunze, Mike Kupfer, and James G. Thompson. A trace-driven analysis of the UNIX 4.2 BSD file system. *Proceedings of 10th ACM Symposium on Operating Systems Principles* (Orcas Island, Washington). Published as *Operating Systems Review*, 19(5):15–24, December 1985.
- [Patterson88] David A. Patterson, Garth Gibson, and Randy H. Katz. A case for redundant arrays of inexpensive disks (RAID). *Proceedings of SIGMOD* (Chicago, Illinois), 1–3 June 1988.
- [Reddy91] A. L. Narasimha Reddy and P. Banerjee. Gracefully degradable disk arrays. *Proceedings of FTCS-21*, pages 401–8. Institute of Electrical and Electronics Engineers, June 1991.
- [Ritchie84a] D. M. Ritchie. The evolution of the UNIX time-sharing system. *AT&T Bell Laboratories Technical Journal*, 63(8, part 2):1577–93, October 1984.
- [Rosenblum92] Mendel Rosenblum and John K. Ousterhout. The design and implementation of a log-structured file system. *ACM Transactions on Computer Systems*, 10(1):26–52, February 1992.
- [Ruemmler93] Chris Ruemmler and John Wilkes. UNIX disk access patterns. *Proceedings of Winter 1993 USENIX* (San Diego, CA), pages 405–20, 25–29 January 1993.
- [Ruemmler94] Chris Ruemmler and John Wilkes. An introduction to disk drive modeling. *IEEE Computer*, 27(3):17–28, March 1994.
- [Saltzer81] J. H. Saltzer. End-to-end arguments in system design. *Proceedings of 2nd International Symposium on Operating Systems* (Paris), April 1981.
- [Schulze89] Martin Schulze, Garth Gibson, Randy Katz, and David Patterson. How reliable is a RAID? *Spring COMPON'89* (San Francisco), pages 118–23. IEEE, March 1989.
- [Stodolsky93] Daniel Stodolsky, Garth Gibson, and Mark Holland. Parity logging: overcoming the small write problem in redundant disk arrays. *Proceedings of 20th International Symposium on Computer Architecture* (San Diego, CA), pages 64–75, 16–19 May 1993.
- [Wilkes91] John Wilkes and Raymie Stata. Specifying data availability in multi-device file systems. *Position paper for 4th ACM-SIGOPS European Workshop* (Bologna, 3–5 September 1990). Published as *Operating Systems Review*, 25(1):56–9, January 1991.
- [Wilkes95] John Wilkes, Richard Golding, Carl Staelin, and Tim Sullivan. The HP AutoRAID hierarchical storage system technology. *Proc 15th ACM Symposium on Operating Systems Principles* (Copper Mountain Resort, CO). Published as *Operating Systems Review*, 29(4), 3–6 December 1995.
- [Worthington94a] Bruce L. Worthington, Gregory R. Ganger, and Yale N. Patt. Scheduling algorithms for modern disk drives. *Proceedings of ACM SIGMETRICS Conference on Measurement and Modeling of Computer Systems* (Nashville, TN), 16–20 May 1994.

Author information

John Wilkes graduated with degrees in Physics (BA 1978, MA 1980), and a Diploma (1979) and PhD (1984) in Computer Science from the University of Cambridge. He has worked since 1982 as a researcher and project manager at Hewlett-Packard Laboratories. His main research interest is in fast, highly available, distributed storage systems, although he also dabbles in network architectures (the Hamlyn sender-based message model), OS design (most recently in the Brix project), harassing PhD students at random universities whenever he gets the chance, and learning about early Renaissance art and architecture.

Stefan Savage graduated with a degree in Applied History (BS 1991) from Carnegie Mellon University. He worked there for three years developing real-time operating systems and then left for Seattle, where he learned to like coffee and became a graduate student at the University of Washington. His current research interests are in safe extensible operating systems (such as SPIN, which has been occupying most of his time for the last two years) although he also finds time to be harassed by visiting researchers from random industrial research labs. He prefers art with an explanation.

The authors can be reached by electronic mail at savage@cs.washington.edu and wilkes@hpl.hp.com.

A Comparison of OS Extension Technologies

Christopher Small and Margo Seltzer
Harvard University

Abstract

The current trend in operating systems research is to allow applications to dynamically extend the kernel to improve application performance or extend functionality, but the most effective approach to extensibility remains unclear. Some systems use safe languages to permit code to be downloaded directly into the kernel; other systems provide in-kernel interpreters to execute extension code; still others use software techniques to ensure the safety of kernel extensions. The key characteristics that distinguish these systems are the philosophy behind extensibility and the technology used to implement extensibility. This paper presents a taxonomy of the types of extensions that might be desirable in an extensible operating system, evaluates the performance cost of various extension technologies currently being employed, and compares the cost of adding a kernel extension to the benefit of having the extension in the kernel. Our results show that compiled technologies (e.g. Modula-3 and software fault isolation) are good candidates for implementing general-purpose kernel extensions, but that the overhead of interpreted languages is sufficiently high that they are inappropriate for this use.

1 Introduction

The motivation that led to the emergence of microkernels in the early 80's leads now to the emergence of extensible operating systems. It is unreasonable to expect any one system to possess all of the functions needed by all applications. Rather, vendors of sophisticated applications, which require application-specific operating system customization, sell these extensions as well. These kernel extensions enable new functionality, but they also introduce the possibility of compromising system reliability. If an application consistently brings a system down, its additional functionality is hardly worthwhile. Therefore, one of the major challenges that arises in supporting easily

extensible operating systems is being able to do so in a manner that does not compromise the reliability of the underlying system.

Commercial systems have traditionally taken the route of supporting dynamically loadable extensions (e.g. device drivers, new subsystems, and new file systems). This is a practical approach, but does not address the reliability issue; it is assumed that it is acceptable for the kernel to crash if the module has bugs in it and if the module does not contain security violations. This scenario makes sense when a small group of trusted users are the only ones who can load such extensions, but a general purpose facility must be more robust. Several current research projects are developing more general-purpose extensible systems that provide reliability and security. The goal of this paper is to describe the assortment of technologies available and the types of extensions one might envision, and then to assess the performance trade-offs of the various choices.

In Section 2 we discuss several of the extensible systems currently under development. Section 3 describes the different types of kernel extensions that might be useful. Section 4 discusses the various extension technologies, and section 5 introduces our test cases and presents our performance results and analysis.

2 Related Work

If one chooses to ignore the potential reliability problems introduced by adding code to the kernel, the no-overhead solution is to allow unprotected patching. Microcomputer operating systems (e.g. MS-DOS and the Macintosh OS) allow arbitrary code to be patched into the kernel, providing no protection from misbehaved applications. On the other hand, because applications have complete control, it is much easier to manipulate and extend the kernel to provide for the needs of applications. It may be easier to write a real-time system on a microcomputer operating system than on a workstation operating system *because* the kernel lacks multitasking and protection.

The move towards safe extensible operating systems is an evolutionary step on the path from

This research was supported by Sun Microsystems Laboratories.

conventional monolithic kernels to microkernel architectures. The Mach microkernel [ACCE86] was designed to allow kernel functionality to be moved out of the kernel address space into user-level external servers in order to increase safety, robustness, and flexibility. However, the expense of frequent upcalls to user-level code motivated the current generation of microkernel-based systems, which link server code directly into the kernel address space [GUIL91]. The CMU Bridge project follows this model, placing servers in the kernel, but protects the kernel using software fault isolation techniques such as *sandboxing* [WAHBE93]. Measurements of sandboxing have shown it to have a much lower overhead than hardware protection mechanisms (on the order of a few percent).

The Exokernel project [ENGL95] also strives to reduce the number of user-kernel protection boundary crossings, but it takes a different approach. Rather than support safe downloading of code into the kernel, it moves as much functionality as possible from kernel to user-level. In this reductionist approach to kernel design, kernel abstractions are thought to be the reason that systems have poor performance, hence they are removed from the kernel and placed into the application. System abstractions are available from user-level libraries, but applications control which abstractions they use. The kernel is left only with responsibility for controlling access to physical devices. There is little or no reason to extend the kernel; nearly all functionality is under the control of the application.

Along with adding functionality, extensible systems can provide applications with the ability to override policy decisions. Cao et al. [CAO94] motivate this sort of extension by examining the performance improvement achieved by allowing an application to control the buffer cache eviction policy. Their system did not allow applications to add new policy code to the kernel; rather, multiple policies were compiled into the kernel and an application chose among them. This work showed the benefit of allowing applications to control policy; however, we believe that it is not possible to determine (and implement) all policies *a priori*; a more general extensibility mechanism is required.

The HiPEC system [LEE94] is similar to, but more flexible than, Cao's system. HiPEC allows applications to control VM caching policy using programs written in a simple, assembler-like, interpreted language designed specifically for the task of managing a queue of VM pages. The performance impact of executing a program in this language is low, but the expressiveness of the HiPEC language is

limited (it has only 20 basic instructions). The language would have to be augmented if it were to be used for other applications.

Packet filters are used to demultiplex a stream of network packets by examining the contents of each packet header. Often, packet filters are implemented in a simple interpreted language (e.g. [MOGUL87, MCCAN93, YUHARA94]). A special language, designed to efficiently describe packet headers, is used to write packet filters. The performance of interpreted packet filters is close to that of compiled code, but, like HiPEC, the expressiveness is limited to the specific domain.

Instead of starting with a minimal language and extending it, the SPIN system [BERS95] is written in, and uses as its extension language, Modula-3 [NELS91]. Modula-3 is a strongly-typed, garbage-collected language, designed so that it is impossible for a program to have a "dangling pointer" to deleted data or to construct a pointer to an arbitrary memory location. Because of the design of the language, code in "safe" modules is not able to reach outside its bounds and violate the integrity of the program in which it is running.

The μChoices operating system [CAMP95] proposes using "a simple flexible scripting language similar to Tcl" to aggregate multiple kernel calls or remove control traffic between user-level and kernel-level. Although Tcl was not designed with this application in mind, if extensions are relatively small, the raw performance of the extension technology will be of little or no consequence.

Extension technology is also useful outside the kernel. Some database servers allow clients to load query- or datatype-specific code into the server to improve performance. The Thor database server uses a typesafe language designed for writing extensions [LISK95]; the Illustra database server is extended by writing DataBlades, which add support for new data types to the server [ILLU94]. The Illustra server does not currently protect itself from misbehaved DataBlade code, although Illustra is evaluating software fault isolation techniques.

The HotJava Web browser from Sun Microsystems can be extended with "applets" written in Java, a Modula-3 like language with a C++ syntax [GOSL95]. Java code is compiled to a machine-independent byte-code that is downloaded by the HotJava browser and interpreted or compiled into native code on-the-fly. Like Modula-3, Java is designed to reduce or eliminate dangling or stray pointers and safety problems.

In this work, we examine the extension technologies being proposed by these systems: unsafe

C (the DOS approach), C in user-level servers (the microkernel approach), software fault isolation, Modula-3, Java, and Tcl. Our goal is to understand the performance implications of the extension technology choice.

3 Graft Taxonomy

We refer to a kernel extension as a *graft* and the process of adding a graft to the running kernel *grafting*¹. We have found three motivations for grafting code into the operating system kernel:

- *Policy*: the application wants to control kernel policy, e.g. how it manages the buffer cache, VM cache, or process scheduling.
- *Performance*: the application wants to migrate portions of itself into the kernel in order to improve overall performance. This technique can save data copies between kernel and user space (e.g. when copying data from the disk to the network), and upcalls into user space (e.g. to handle a mouse event).
- *Functionality*: the application wants to add general functionality to the kernel, e.g. support for access control lists, automatically compressed files, or new communication models.

Although there are an unlimited number of ways in which a graft can be structured, we have identified three basic structures into which the implementation for most grafts will fall: *Prioritization* grafts, *Stream* grafts, and *Black Box* grafts.

3.1 Prioritization

There are numerous places in a kernel where one of a set of entities is selected. Choosing a victim is the central policy decision made by the virtual memory system (which page to evict), the buffer cache manager (which buffer to evict), and the process scheduler (which process to schedule next). We call this a *Prioritization* policy decision, and a graft that replaces prioritization policy code a *Prioritization graft*.

Normally, the VM system and the buffer cache use a least-recently-used (LRU) policy (or a variant thereof), although in some cases a different policy works better. An application might know that each block of a file will be read once, in order, and not read again, in which case it makes sense to use a most-recently-used (MRU) strategy for that file. As another example, while processing a query, a database system

knows which blocks of the database will be needed soon and which blocks are no longer needed.

The access pattern of a set of pages or file blocks is often a good heuristic for deciding which pages or blocks will be used in the near future, but it is not infallible. For example, a garbage collector copies live objects from partially filled pages with the goal of decreasing fragmentation. After a collection, a page from which live objects have been taken has been accessed recently, but contains no useful data; on the other hand, pages that are full of live data have not been accessed recently. Using an LRU strategy, the former will be retained in memory and the latter will be evicted, which is the opposite of what is desired.

Process scheduling is another example of a prioritization policy. At each scheduling point the kernel has a list of candidates, and chooses one to run. No scheduling algorithm is appropriate for all application mixes; the demands of interactive applications differ from those of applications with real-time deadlines, and a scheduler optimized for one class will not satisfy the other. Processes may wish to be scheduled as a group; a client-server application may not want the server to be scheduled unless there is an outstanding client request, in which case it should be scheduled ahead of any client.

In general, a Prioritization graft is one that is presented with a list of options and must select the item of highest priority. It uses some internally defined weighting function to choose a candidate. The weighting function may use just the information in the list, or it may have access to some data from the application (e.g. which blocks or pages will not be needed again).

Our Prioritization graft benchmark models a VM page eviction policy. We assume that the kernel maintains a list of pages in LRU order; when it comes time to evict a page, the kernel normally chooses the page at the head of the LRU queue as its candidate. In our model, instead of immediately evicting the candidate, the kernel determines which process owns the candidate page and allows the process to offer one of its other resident pages for eviction. Our model application keeps a “hot list” of pages that will be needed in the near future. The goal of the page eviction graft is to ensure that none of these pages are evicted.

The page eviction graft receives a pointer to the head of the LRU queue. The graft checks to see if the candidate (the head of the queue) is on the application’s hot list; if it is not, it accepts the kernel’s candidate. If the candidate is on the hot list, the graft searches through the queue for an acceptable page

1. This is a (weak) pun on the name of our project, the VINO kernel.

that is not on the application's hot list. This page is returned to the kernel.

For this benchmark we model a TPC-B transaction processing benchmark database [TPCB90]. The database holds 1,000,000 records in a four-level b-tree; the b-tree has approximately 400 internal pages (16MB) and 50,000 data pages (200 MB). The b-tree is 50% full, and has one root page, four pages at the second level, 391 pages at the third level, and approximately 50,000 pages at the fourth level; each third-level page in the b-tree points to up to 128 fourth level pages.

The server accesses the database by mapping it into its virtual address space. When the server does a non-keyed lookup, it traverses the b-tree in depth-first order, starting from the root. When it reaches a third-level page it knows which 128 fourth-level pages it will access next, hence which of the memory mapped pages of the database should not be paged out. During such a search, it constructs a hot list of these 128 pages. If a page fault occurs during a search, our graft is called with the LRU chain head, and it uses the hot list to find an eviction candidate that is acceptable to the application. As each page is processed, its entry is removed from the hot list, so as the simulation runs, the queue grows shorter. We presume that the kernel keeps track of candidate pages and graft-proposed alternates, as in Cao's system [CAO94], to ensure that an application does not manipulate the VM system to gain more physical memory than it would receive under the default strategy.

This test is not particularly compute-intensive. Instead, it is sensitive to the overhead associated with traversing a list of items. If the extension technology requires extensive pointer checking, or does not support traversal of linked lists, this test will highlight it.

Because there are 50,000 data pages in the database, there is a low probability that a page that the application will need is already in the cache (roughly 64/50,000, or once every 781 times). However, if one of the pages is in memory, we want to ensure that it is not evicted. For the graft to be successful, the overhead of checking on each eviction should be low relative to the cost of evicting and then re-faulting a page. Our test computes the break-even point, showing how often the graft will need to save an eviction in order to pay for its per-eviction cost.

3.2 Stream

Our second graft model is based on the idea of Unix filters and pipes. Frequently, it is useful to apply a set

of filters to a data stream. For example, we might want the kernel to transparently compress a file when it is written and decompress it when it is read, or automatically encrypt a file when written and decrypt it when read by the appropriate user. For security reasons we might want to compute a secure checksum, or *fingerprint*, of an executable when it is loaded, to verify that it has not been compromised by a virus. A *stream graft* is such a filter. It consists of filtering code that is inserted into a data stream, normally between the storage system and application level.

A journaling file system is one that accepts a stream of I/O requests, saves a journal of the metadata changes, and passes along the original requests. A standard filesystem could be transformed into a journaling filesystem by inserting into the request stream a graft that journals the changes made to the metadata.

The Stream Input-Output System of UNIX [RITCH84] decomposed the character I/O system of UNIX into a set of filters. Network and terminal protocols were built up by linking filters into chains. Characters read from (or sent to) a device were passed to the first filter in the chain; each filter processed the characters in its input queue and moved them on to the next filter in the chain. This mechanism was used not only to handle erase and kill processing, but also to create pseudo-devices, such as virtual displays and keyboards, to construct multiple virtual terminals from a single terminal.

Another example of a stream graft is one that takes a data source (e.g. the disk) and, rather than modifying it on the way to application level, writes it elsewhere (e.g. the network). Recent research into fast path connections, such as the *x*-kernel work at Arizona [DRUS93], the video server benchmark of the SPIN operating system [BERS95], and Fall's work in decreasing I/O time through use of in-kernel copying [FALL93] show that there is a substantial performance gain from saving copies to and from user-level. A stream graft that takes its input and directs it to an output connection, perhaps after transforming the data, could be used to build this type of fast path connection.

Our representative stream graft is an implementation of the MD5 Message-Digest Algorithm [RFC1321], which produces a 128-bit fingerprint of a file. The MD5 fingerprint is both expensive to compute and computationally infeasible to forge. MD5 is useful for ensuring that a file has not been tampered with; a change to the contents of the file will result in a change to the fingerprint. If the fingerprint is kept separate from the file (say, on a

small amount of safe media, such as a read-only floppy disk), a change to the file can be detected by computing its MD5 fingerprint and comparing it to the saved fingerprint.

Like many compression and encryption algorithms, MD5 is stream-based. The algorithm maintains a small amount of state as it processes the data; unlike compression and encryption algorithms, the data output is the same as the input; when the algorithm completes, the graft can be queried for the fingerprint, and the computed fingerprint can then be compared to the saved fingerprint.

If the MD5 fingerprint can be computed as quickly as data can be read from the disk, the time spent in the MD5 code can be overlapped with I/O activity. However, if it takes longer to compute a fingerprint than it takes to read the data from the disk, the overall processing time will increase. Our test measures whether a graft written in a given technology can keep up with the disk.

3.3 Black Box

The Black Box graft structure is more general than that of a Prioritization or Stream graft. A Black Box graft has some number of inputs, some state, and a single output. We see it operating as a “black box” function, normally producing a single output value. For example, at the center of the code that implements Access Control Lists is a small database that (at an abstract level) accepts a triple containing a file access request, a user ID, and a file ID, and responds “yes” or “no.”

File system read-ahead code, which determines how many (and which) blocks of a file to prefetch, is another example of a “black box” function. If the application knows ahead of time the order in which blocks of a file will be read, the kernel can use this information to make read-ahead decisions. In some cases, an application will read a subset of the blocks of a file in order, and then skip to another region of the file. If the kernel uses heuristics (rather than application knowledge) to choose a read-ahead policy, it can not cope with arbitrary application behavior. With the cooperation of the application, it can make more appropriate read-ahead decisions.

A Logical Disk facility (LD) [DEJON93] sits between the filesystem and the physical disk. The filesystem reads and writes logical blocks, and the LD maps the logical requests to locations on the physical disk. The LD can be used to transparently replicate data, by writing it in multiple places on the same disk or multiple disks, and speed write performance, by

writing logically discontiguous blocks on a physically contiguous region. A log-structured file system [ROSE91] can be implemented using a logical disk facility; the filesystem lays out blocks as it sees fit, and the Logical Disk reorders and buffers writes to improve write performance.

Our black box test application is a simple logical disk facility that converts random writes to sequential writes. It accepts block write requests, batches them into physical segments, and maintains a mapping from logical block numbers to physical block numbers. If the savings in I/O time due to batching is greater than the cost of translating logical block numbers on each read/write, the graft is effective.

4 Extension Technologies

Each extension technology offers a different level of safety and imposes a different level of trust. An extension written in an unsafe language (e.g. C) can read, write, or jump to any location in the address space (using pointer arithmetic); one written in a safe language (e.g. Modula-3 or Java) is restricted to code addresses exported to the extension. In either case, we need a mechanism to ensure that extension code not monopolize the CPU; we must be able to preempt an extension that runs too long. This means that extension code should not be able to disable interrupts.

For an operating system extension, read protection is important, even if it is not required for reasons of data privacy. Because an extension running in the kernel has access to memory mapped devices, an extension can destructively *read* a device register.

Once safety is ensured, the performance of a technology determines its suitability for different applications. The overhead of a graft intended to increase performance should not cause performance to degrade.

We also need to make sure that an extension technology is sufficiently expressive, so that it is possible to implement the grafts we want to write. A small, specialized language designed around a single problem domain may perform better than a general-purpose language, but will be difficult to reuse in other domains.

The size of the runtime environment can have an impact on overall performance. The runtime code size of support environments ranges from tens of kilobytes to several megabytes. If we require that the support environment run in the kernel and remain resident, any memory used by the environment is memory unavailable for other uses.

The extension technologies we examine fall into three basic trust models: *hardware protection*, *software protection*, and *interpretation*.

4.1 Hardware Protection

The simplest, and perhaps most dependable, method for ensuring the safety of the kernel is to place extensions outside the kernel's address space in order to take advantage of the protection offered by the hardware. When the kernel wants to run an extension, it upcalls into user-level code; when the code returns, the kernel continues. Along with memory protection, the kernel can time-slice the extension to ensure that it does not monopolize the CPU. If the extension runs too long, the kernel can abort it and carry on without it.

The primary disadvantage of this model is that there is a cost associated with an upcall; for small extensions, this per-invocation overhead can be much larger than the cost of running the extension.

4.2 Software Protection

Using software protection, we place extension code in the same address space as the kernel, but restrict the instructions that are evaluated by the extension. By controlling the language (e.g. Modula-3), or the compiler, or by patching the binary code [WAHBE93], we can control the instructions evaluated by the extension. This allows us to ensure that the extension does not read or write outside its bounds, or jump to arbitrary kernel code. In addition, we can ensure that it does not issue instructions that disable interrupts or that initiate I/O operations.

Software protection can offer the highest performance of the three options, but requires control of, and trust in, the language translation tools used to process the extension source. The security of the kernel is a function of the correctness of the generated code; if the translation tool can be coaxed into generating unsafe code, the security of the kernel can not be guaranteed. If, for example, under certain circumstances the array bounds checking of a compiler can be subverted, it may be possible to write code that overwrites its stack and circumvents the kernel's security mechanisms.

Even assuming that the translation tools work correctly, the kernel still needs to verify that code loaded into the kernel was in fact processed by a trusted language tool. This can be accomplished by performing the translation at load time, or by marking the code (say, with a cryptographic checksum) at the

time it is generated. The former technique requires a high cost at extension load time; the latter implies that we are able to embed a secret key of some sort in the language tool and mark the generated code with this key. Methods for generating cryptographic checksums are well known; the social issues of key distribution and management are outside the scope of this paper.

The goal of *software fault isolation* (SFI) is to make it more efficient to ensure the validity of memory references using software than by using hardware. One type of SFI, *sandboxing*, ensures that the high bits of a memory address match those of the *sandbox* region assigned to the function or module. In this way, a module can, at worst, overwrite its own data with a stray pointer or jump to locations in its own code. Sandboxing can be done at compile time or performed as a post-processing phase on object files, allowing separation of the sandboxing tools from the compiler. At load time, a linear-time algorithm can be used to guarantee that all memory references in a piece of object code have been correctly sandboxed.

Omniware C++ [COLU95] is a commercial system that includes a compiler that generates machine independent code. A run-time system translates the machine independent code into native code with software fault isolation instructions and links the code into the executable.

Many modern languages, such as Modula-3 and ML, do not suffer the safety problems of C. In a *typesafe* and *pointer-safe* language it is impossible to construct a pointer to an arbitrary memory location, or be left with a "dangling" pointer to deallocated memory. If type casting is available, it is combined with compile-time analysis or a runtime type check to ensure that the cast is valid. Similarly, array bounds are checked on access to ensure that a program does not access memory outside the array.

Although the definition of a language may ensure type and pointer safety, the language tools written to compile and run the language may not correctly implement the definition. It is infeasible (using present technology) to verify the correctness of a Modula-3 compiler; because of this, we can not be sure that a compiler will not generate incorrect code. (In fact, while running our VM Page Eviction benchmark, we found a bug of this type in the Modula-3 compiler.)

4.3 Interpretation

Instead of depending on the safety and security of a compiler, we can create a virtual machine, an interpreter, to run grafts. The source language (which

could be C, Modula-3, or anything else) would be compiled to intermediate or machine code for a real or virtual machine. The kernel would include an interpreter to run the code, ensuring its safety. This model allows complete control over the behavior of the extension by implementing only safe operations in the interpreter.

The intermediate code could also be used as the input to a runtime code generator. A reasonably fast interpreter runs 10 to 100 times more slowly than compiled code [MAY87]; however, using incremental code generation techniques, performance can approach that of compiled code [HOLZE94]. (Note that there is a flexible line between generating native code at load time – as above – and dynamically generating native code from interpreted code.)

Java is compiled to a compact byte code for the Java Virtual Machine. As interpreters go, the Java interpreter is fairly fast. In addition, Sun plans to release a runtime code generator for Java in the near future. The authors of Java expect that compiled Java will run at about the same speed as compiled C or C++ ([GOSL95], p. 48), which, based on current technology, is believable.

The currently available Java system is Alpha-release software. Versions for Sparc/Solaris and Windows are available from Sun (on java.sun.com); Java has been ported to HP-UX and UnixWare (by OSF) and Linux (found on java.blackdown.org).

Another technique for building an interpreted language is not to transform the source to an intermediate format, but rather to interpret it directly. This technique, used in awk, sh, and Tcl, leads to a smaller start-up time, with a higher overhead per statement. Because source-interpreted scripting languages are immensely popular, and have been proposed as a vehicle for writing grafts [CAMP95], we include Tcl as one of our tested technologies.

5 Performance Analysis

Given the wide range of extension technologies available, it is not obvious which is “best” in any dimension. In fact, extensible systems are being built that employ nearly every technology described. In this section, we measure and analyze the performance of our sample extensions.

5.1 Hardware

We ran tests on four hardware platforms: three commercial workstations and one Intel x86 “PC”.

- *Alpha*: DEC AlphaStation 400 4/233 (233MHz), running DEC OSF/1 v3.2A, 64MB of memory.

- *HP-UX*: HP PA-RISC 9000/735 (99MHz), running HP-UX A.09.03, 80MB of memory.
- *Linux*: “PC”-class Pentium (90MHz), running Linux 1.1.95, 16MB of memory.
- *Solaris*: Sun SPARCStation 20 (75MHz), running SunOS 5.4, 128MB of memory.

5.2 Extension Technologies

We implemented our grafts on five different extension technologies. Not all tests were run on all platforms; for example, the current release of the Omniware compiler runs only on Solaris.

- *C*: compiled with gcc -O (version 2.6.3 on HP-UX, 2.7.0 on all other platforms).
- *Java*: release Alpha 3.
- *Modula-3*: version 3.5.3 of DEC SRC Modula-3.
- *Omniware*: compiled with Colusa’s omniC++ compiler, version 1.0 beta, release 1.5. (This commercially available compiler generates machine-independent code that is translated to native, fault-isolated code at runtime. This pre-release version of the compiler supports write and jump protection, but no read protection, and does not include an optimizer for the SFI instructions.)
- *Tcl*: Tcl version 3.7.

5.3 Upcall Overhead

As a baseline for comparison, we implemented each benchmark in C. Since C is not a safe language, we estimated the cost of implementing the benchmark as a user-level server by determining the break-even point as a function of the time required to upcall to such a server; if an upcall is free (takes no time), the performance of a system written using user-level servers is equal to that of one with unsafe C linked into the kernel. As the cost of an upcall increases, the performance of a system written using user-level servers will decrease.

When the kernel makes an upcall to a user-level server, it pushes a new call frame on the server’s stack and switches to the server. When the server is done handling the upcall, it returns to the kernel. This process is similar to (but simpler than) the one followed by the kernel when it posts a signal to a process. First, the kernel pushes a call frame for the signal handler on the application’s stack, then schedules the application process. When the process returns from the signal handler, it re-enters the kernel. The kernel cleans up the state of the process and returns it to the point where the signal was handled.

To give a feeling for the cost of an upcall on each platform, we measured the time required to handle a signal sent to a process. The test program forks a child process, which registers handlers for a group of twenty signals and then suspends itself (by sending itself SIGTSTP). When the parent is notified that the child is suspended, it posts the handled signals to the child, then wakes it (by sending it SIGCONT). The child awakes, handles the signals, and suspends itself again. When the parent is notified that the child has once again been suspended, it knows that all signals have been handled.

We then measured the time to post the signals to the child when the child ignores (rather than handles) the group of signals. The latter time is subtracted from the former; the result is divided by the number of signals handled, which gives an estimate of the time required to handle a single signal. The results of this test are shown in Table 1.

Platform	Signal Handling Time
Alpha	19.5 μ s(7.5%)
HP-UX	25.8 μ s(1.4%)
Linux	55.9 μ s(0.1%)
Solaris	40.3 μ s(3.8%)

Table 1. Signal Handling Time We measure the time required to send twenty signals to a child process that handled the signals, then subtract the time required to send twenty signals to a child process that ignores the signals. The difference is divided by the number of signals to give a per-signal handling time. Each time is the mean of thirty runs of 1000 iterations each (standard deviations in parenthesis).

We implemented and measured the performance of a simple upcall mechanism on BSD/OS 2.0. On a 486-DX266 we measured a signal handling time of 63.1 μ s, and an upcall time of 37.2 μ s (about 40% quicker).

These upcall estimates are fairly conservative. Work done to improve exception handling times ([THEK94]) and cross-domain procedure calls ([Ford94]) lead us to believe that it is feasible to implement an upcall mechanism that takes less than one fourth the time we measured for signal delivery.

5.4 VM Page Eviction

As described in Section 3.1, our sample Prioritization graft takes the LRU-ordered list of page eviction candidates and returns a candidate not on its hot list.

We assume that the application keeps the hot list (of active pages) in its memory at a known location, so that it is accessible to the graft. In addition, we

assume that the application keeps the hot list in a form that can be easily traversed by the graft; for example, the C graft searches a linked list of structs, where the Modula-3 graft searches a linked list of Modula-3 RECORDS.

In our model application, the hot list starts out with 128 entries (the number of data pages referenced by a level-three internal page), and as each page is faulted, it removes that page from the hotlist. On average, the hotlist contains 64 pages, which is the number we simulate.

We determined the time to check the 64 element hotlist in each of the supported technologies, and present that time in Table 2. To give an intuitive feeling for the performance of each technology, we also normalized the time relative to unsafe C code.

To determine the break-even point for this graft, we measured the page fault time on each of our test platforms (Table 3). The times listed indicate how long it takes to handle a page fault event, not how long it takes to bring in a single faulted page; Alpha and HP-UX bring in multiple disk pages on each fault.² We assume that the systems are performing read-ahead in order to take advantage of (expected) locality of reference. However, our model database server would not be able to take advantage of this behavior, as the faulted data pages are scattered throughout the database.

(The page fault read-ahead policy exhibited here is an obvious candidate for grafting; if we are able to control how many pages the system brought in on a fault, we can reduce the per-fault time.)

Once we have computed the page fault time, we can determine the break-even point for this graft. We divide the page fault time by the time required to run the graft; the result is the number of times we can run the graft for each page eviction saved and still be ahead of the game.

Remember that our model application would find a page to save, on average, once every 781 invocations. If the break-even point is less than this, our model application would not benefit from this graft. Worse yet, if the number is less than one, the amount of time to run the graft is greater than the page fault time, hence under no circumstances would the graft be beneficial.

In Table 2 we see that on Solaris the relative times of Omniware and Modula-3 are quite close, each running about 40% slower than unprotected C.

2. The number of pages faulted was determined by running the page fault test on an otherwise unloaded system while watching the output of iostat or vmstat.

Platform		C	Java	Modula-3	Omniware
Alpha	raw	2.9 μ s(0.2%)		3.2 μ s(1.5%)	
	normalized	1.0	N.A.	1.1	N.A.
	break-even	8655		7843	
HP-UX	raw	6.0 μ s(0.4%)	159 μ s(0.8%)	6.8 μ s(1.7%)	
	normalized	1.0	26.5	1.1	N.A.
	break-even	2983	113	2632	
Linux	raw	3.7 μ s(0.1%)	237 μ s(0.1%)	9.1 μ s(0.1%)	
	normalized	1.0	64	2.5	N.A.
	break-even	1270	20	516	
Solaris	raw	4.5 μ s(0.1%)	141 μ s(0.3%)	6.3 μ s(2.8%)	6.3 μ s(0.2%)
	normalized	1.0	31.3	1.4	1.4
	break-even	1533	49	1095	1095

Table 2. VM Page Eviction Test We measure the mean time required to search a 64 element “hot list” of page numbers. Raw times and time normalized to unprotected C code (on the same platform) are given. The break-even point is the number of times the graft can run in the time it takes handle a page fault. Each time is the mean of 30 runs of 100,000 searches each (standard deviations in parenthesis).

(Remember, however, that this version of Omniware does not include read protection, which gives it a performance advantage over Modula-3.) On Alpha and HP-UX the Modula-3 code runs 10% slower than the C code, which is not a significant difference for this test.

Platform	Fault Time	Num Pages
Alpha	25.1ms(5.0%)	16
HP-UX	17.9ms(0.8%)	4
Linux	4.7ms(0.5%)	1
Solaris	6.9ms(3.2%)	1

Table 3. Page Fault Time Measured using *lmbench* (standard deviations in parenthesis). Alpha and HP-UX bring in more than one disk page on a fault, performing read-ahead, even though the test performs random accesses to memory.

On Linux, we see a 150% slowdown for Modula-3, a greater difference than we see on other platforms and with other tests. Examining the code generated by the Modula-3 compiler, we found that it includes a runtime check against NIL (location zero) on each pointer access. The code generated on the other platforms (Solaris, Alpha, and HP-UX) does not include explicit NIL checks. The Modula-3 language specification [NELS91, p. 50] states that dereferencing NIL should cause a runtime error; on Solaris and Alpha, dereferencing location zero causes a segmentation violation, which is trapped by the Modula-3 runtime system. This is not the case on Linux, so the runtime check is needed. (We found that

although no runtime checks are generated by the HP-UX version of the Modula-3 compiler, dereferencing location zero does not cause a segmentation violation. This appears to be a bug in the Modula-3 compiler.)

For our purposes (i.e., operating system extensions), the Alpha and Solaris slowdowns are the more appropriate comparison – because we can ensure that dereferencing location zero causes a fault, we would not need runtime NIL checks. (This fault would not be handled by a signal mechanism in the kernel, but would instead require some support by the normal kernel fault logic.)

On Solaris, the Java code runs at about 1/30th the speed of compiled C code, and about 1/20 of Modula-3. On this platform we see that the break-even point for Modula-3 is about 1100 pages; for Java, the break-even point is about 50 pages, too low to benefit our model application.

To compare these results to the overhead of performing an upcall on each eviction, we computed the break-even point as a function of upcall time, allowing the upcall time to range from 0 to 50 μ s (see Figure 1). When compared with the break-even points for Modula-3 and Omniware, we find that we would need an upcall time on the order of 5 μ s for upcalls to compete with the compiled extension technologies, which would be difficult to achieve.

Unsurprisingly, we found that Tcl is not the appropriate tool for this job. Measurements of Tcl showed that it was four orders of magnitude slower than the compiled languages (C and Modula-3), taking 40ms on Solaris, as compared with the C version, which took 4.5 μ s. With a break-even point

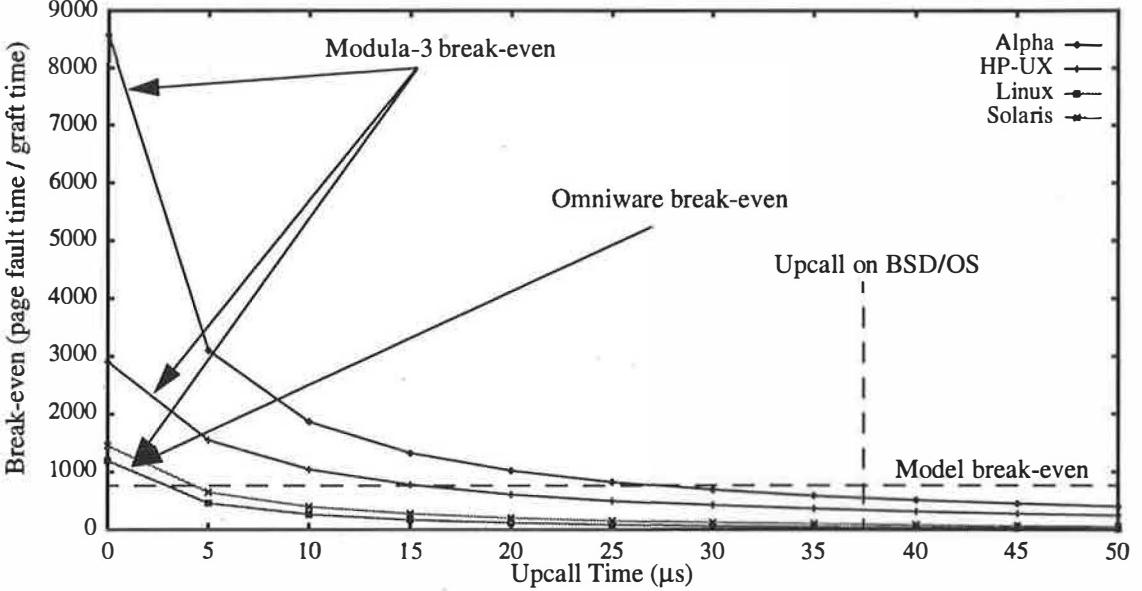


Figure 1. Break-Even vs. Upcall Time The break-even point for the VM Page Eviction test. Break-even is inversely proportional to the upcall time. The break-even points for Modula-3 and Omniware are included, showing that a sub-10μs upcall time is needed for user-level servers to compete with compiled, downloaded code here.

close to or less than one (i.e. the Tcl code would have to save a page each time it was called in order to not degrade performance), Tcl is not competitive here with the other technologies.

5.5 MD5 Fingerprinting

We took the standard C implementation of the MD5 algorithm (found in RFC1321) and modified or reimplemented it for each of our test platforms. The MD5 algorithm makes heavy use of array access and unsigned 32-bit arithmetic. The standard implementation takes advantage of C's behavior of silently ignoring numeric overflow by performing computation modulo 2^{32} . The code worked correctly in C on all platforms, and was easily translated into Java and Modula-3.

This test is very compute intensive. It uses almost none of the support facilities of the extension technology (e.g. no data types other than array), but instead gives an indication of the cost of performing array access and numerical computation. The results are shown in Table 5.

Because the cost of computing the fingerprint is so high, the cost of an upcall is comparatively low. If we assume that there would be one upcall to the user-level extension for every 64KB read from disk, we would need to add the cost of $1\text{MB}/64\text{KB} = 16$ upcalls. Assuming a (pessimistic) upcall time of 50μs, this would add 8ms to the raw C time, which is not

significant when added to compute times of 150ms to 800ms.

To compute a break-even point, we measured the write bandwidth of the disk on each system (Table 4). This measurement was used to estimate the time required to read 1MB of data. If the time needed to compute the checksum of 1MB of data is less than that required to read the data from the disk, computation can time can be hidden by I/O time. (This optimistically assumes that the disk read requires no CPU cycles, and is a best-case break-even calculation.)

Platform	Bandwidth (KB/s)	1MB access time
Alpha	4364(1.2%)	235ms
HP-UX	1855(13%)	552ms
Linux	1694(5.7%)	604ms
Solaris	3126(11%)	320ms

Table 4. Disk I/O Time Write bandwidth in KB/s on each platform, measured using *lmbench*. From this, the time to access 1MB of data is computed. Each time is the mean of 30 runs (standard deviations in parenthesis).

The Omniware code is faster than the Modula-3 code, but slower than compiled C. To see if the size of the test was affecting the results, we ran a larger test (64MB) and saw a similar overhead (14480ms for Omniware vs. 9498ms for C, an overhead of 50%). We believe that this overhead is representative of this implementation of Omniware for MD5. (Once again,

Platform		C	Java	Modula-3	Omniware
Alpha	raw	159ms(1.8%)		207ms(0.4%)	
	normalized	1.0	N.A.	1.3	N.A.
	MD5/disk	0.67		0.9	
HP-UX	raw	239ms(1.6%)	23987ms(2.5%)	352ms(0.3%)	
	normalized	1.0	100	1.5	N.A.
	MD5/disk	0.43	43	0.64	
Linux	raw	202ms(0.3%)	22887ms(1.0%)	387ms(0.1%)	
	normalized	1.0	113	1.9	N.A.
	MD5/disk	0.33	38	0.64	
Solaris	raw	146ms(1.7%)	10368ms(0.3%)	294ms(0.1%)	219ms(0%)
	normalized	1.0	71	2.0	1.5
	MD5/disk	0.46	32	0.92	0.68

Table 5. MD5 Fingerprinting Mean time required to compute the MD5 fingerprint of 1MB of data. The time is compared to the time needed to read 1MB from the disk. If this number is less than one, the computation of the fingerprint can be overlapped with I/O. If it is greater than one, computing the fingerprint will decrease throughput. The mean of 30 runs is reported (standard deviations in parenthesis).

this version of Omniware does not include read protection, which gives it a performance advantage over Modula-3.)

Unlike C, Modula-3 does not ignore arithmetic overflow. The Word package supports computation modulo the native wordsize, so on the 32-bit platforms (HP-UX, Linux, and Solaris) we were able to implement MD5 efficiently. On the 64-bit Alpha, the Word package performs computation modulo 2^{64} , which produces incorrect results for MD5. We implemented two versions of MD5 on Alpha: one uses 64-bit integers and the Word package, doing roughly the same amount of work as the 32-bit implementations (but computing an incorrect checksum), and the other using 32-bit integers, which produces the correct checksum, but generates more instructions (by a factor of four). For our performance measurements we used the 64-bit version, which gives a more accurate comparison of the relative performance of the technologies. We found that the 32-bit version took approximately ten times as long to run as the 64-bit version. (We see this as an artifact of the compiler implementation, and not a characteristic of the Alpha processor or of Modula-3.)

The Modula-3 code runs from 1/2 to 3/4 the speed of compiled C. There is no reason for the numerical computation to be slower; we attribute the difference to run-time array bounds checking. On all platforms, the time required to compute the fingerprint was less than the time to read the data from the disk, so a Modula-3 implementation of MD5 could keep up with disk access and overlap its computation with I/O time.

On the other hand, we found that neither Java nor Tcl were able to keep up with the disk. The Java code, at best 1/30th disk speed, seems unlikely to be used for this application. And, as above, we found that our Tcl implementation was four orders of magnitude slower than one written in a compiled language, and hence too slow for this type of graft. (On Solaris it took 50 minutes to complete, as compared with 1.9 seconds for the C code.)

5.6 Logical Disk

Our simulation models a logical disk designed to support a log-structured layer between a filesystem and the physical disk. The simulation accepts write requests for logical blocks and maintains the mapping between these logical blocks and the physical blocks onto which they are stored. As with the system implemented by de Jonge et al. [DEJON93], our simulation maintains all data structures in main memory.

We simulate a 1GB physical disk with 4KB blocks and 64KB (16 block) segments. Our simulation uses a stream of block write requests that are skewed so that 80% of the requests are for 20% of the blocks. Because our simulation does not include a cleaner, we run it for 262144 iterations (the number of blocks on the disk).

We measured the absolute time required to maintain the mapping between logical and physical blocks for the entire run. To justify using this graft, it must save more time than it takes: the overhead incurred per write should be less than the time saved

Platform		C	Java	Modula-3	Omniware
Alpha	raw	0.74s(1.9%)		1.3s(2.0%)	
	normalized	1.0	N.A.	1.75	N.A.
	per block	2.8μs		5.0μs	
HP-UX	raw	1.3s(1.3%)	32.2s(0.6%)	2.1s(0.7%)	
	normalized	1.0	25	1.6	N.A.
	per block	5.0μs	123μs	8.0μs	
Linux	raw	1.3s(0.8%)	46.5s(0.1%)	1.7s(0.9%)	
	normalized	1.0	36	1.3	N.A.
	per block	5.0μs	177μs	6.6μs	
Solaris	raw	1.9s(0.2%)	24.6s(0.4%)	2.9s(0.4%)	2.2s(0.1%)
	normalized	1.0	13	1.5	1.16
	per block	7.2μs	94μs	11.1μs	8.4μs

Table 6. Logical Disk Time to handle bookkeeping for 262,144 writes to a Logical Disk. The time is normalized to compiled C code. The per-block overhead is how much time must be saved on each write in order for the graft to break even. The mean of 30 runs is reported (standard deviations in parenthesis).

by batching writes into segments. We found that the compiled technologies (Omniware and Modula-3) add a sub-10μs overhead per write, which is on the order of 1% of a typical disk seek time.

The Java overhead is on the order of 100μs, roughly 10% of a typical seek, so we would need to save one seek for every ten blocks written. This is not an unreasonable assumption to make; interpreted Java would work for this task. (Because of performance of Tcl on the first two tests, we did not take Tcl measurements for this test.)

When looking at a user-level server for managing the mapping, we assume that there is one upcall on each block write, with a upcall estimate of 10μs. This would double the overhead per write, but still keep it substantially lower than that of interpreted Java. The performance of a user-level server for this task would be relatively close to that of compiled code.

6 Conclusions

We believe that our taxonomy of grafts and of graft architectures encompasses a significant share of the operating system extension space. Our sample grafts (VM page eviction, MD5 fingerprinting, and Logical Disk) are equivalent in structure and performance characteristics to the policy, performance, and functionality grafts we envision. Given the performance of these representative grafts, we are able to determine what kinds of extensions are worth building.

The anecdotal evidence has been that structuring a system with fine-grained user-level extensions and upcalls is not feasible; our VM page eviction test

supports this position. On the other hand, the coarse-grained and computationally expensive MD5 test shows that there are some situations where this overhead does not matter, and the Logical Disk simulation shows that upcalls can be successfully hidden in the face of a large number of I/O operations.

Given that we want both safety and performance, a compiled technology such as Modula-3 or SFI is our best choice. Because Modula-3 is unfamiliar to many developers, it may be that a hybrid language with the syntax of C or C++ but the safety of Modula-3 would be more widely accepted. The two compelling candidates are compiled Java and SFI with full (read, write, and jump) protection. Neither is available today, but both are currently under development. In the near future we will be able to measure their performance and compare them directly.

In their current state, neither of the interpreted technologies are up to the task. While Tcl and Java are well suited for interactive applications, where the relevant metric is human perceptual time, they are not suitable for building kernel extensions because system events occur at a finer timing granularity.

7 Status and Availability

All code is available on the World Wide Web, at <http://www.eecs.harvard.edu/~chris>.

Acknowledgments

The authors are grateful to the reviewers for providing valuable feedback. David Black of OSF, the shepherd, was invaluable in helping with the paper, especially in coming up with the scheme for measuring signal delivery time. Special thanks to OSF for allowing us to use a pre-release copy of their port of Java to HP-UX. David Holland kindly lent us his Linux box (and all its free disk space) to obtain the Linux measurements. Dawson Engler of MIT made the helpful suggestion of presenting the break-even times as a function of upcall time. And special thanks to Keith Smith and Chris Thorpe for their help brainstorming and polishing the paper.

We are very grateful to Larry McVoy of SGI for use of his *lmbench* suite [MCVOY96]. We used tests *lmdd.c,v 1.12* and *lat_pagefault.c,v 1.2*.

Bibliography

- [ACCE86] Accetta, M., Baron, R., Bolosky, W., Golub, D., Rashid, R., Tevanian, A., Young, M., "Mach: a New Kernel Foundation for UNIX Development," *1986 Summer USENIX Conference* (July 1986).
- [BERS95] Bershad, B., Savage, S., Pardyak, P., Sirer, E. G., Fiuczynski, M., Becker, D., Eggers, S., Chambers, C., "Extensibility, Safety, and Performance in the SPIN Operating System," *Proceedings of the 15th SOSP*, Copper Mountain, CO (December 1995).
- [CAMP95] Campbell, R., Tan, S.-M., "μChoices: An Object-Oriented Multimedia Operating System," *Proceedings of HotOS V*, pp. 90–94, Orcas Island, WA (May 1995).
- [CAO94] Cao, P., Felten, E., Li, K., "Implementation and Performance of Application-Controlled File Caching," *Proceedings of the First Usenix Symposium on Operating System Design and Implementation*, pp. 165–177, Monterey, CA (November 1994).
- [COLU95] "Omniware Technical Overview", Colusa Software, <http://www.colusa.com>, (1995).
- [DEJON93] de Jonge, W., Kaashoek, M. F., Hsieh, W., "The Logical Disk: A New Approach to Improving File Systems," *Proceedings of the 14th SOSP*, pp. 15–28, Asheville, NC (December 1993).
- [DRUS93] Druschel, P., Peterson, L., "Fbufs: A High-Bandwidth Cross-Domain Transfer Facility," *Proceedings of the 14th SOSP*, pp. 189–202, Asheville, NC (December 1993).
- [ENGL95] Engler, D., Kaashoek, M. F., and O'Toole, J., "Exokernel: An Operating System Architecture for Application-Level Resource Management," *Proceedings of the 15th SOSP*, Copper Mountain, CO (December 1995).
- [FALL93] Fall, K., Pasquale, J., "Exploiting In-Kernel Data Paths to Improve I/O Throughput and CPU Availability," *1993 Winter USENIX Conference*, pp. 327–334, San Diego, CA (January 1993).
- [FORD94] Ford, B., Lepreau, J., "Evolving Mach 3.0 to a Migrating Thread Model", *1994 Winter USENIX Conference*, pp. 97-114, San Francisco, CA (January 1994).
- [GOSL95] Gosling, J., Gilenton, H., "The Java Language Environment," available from <http://java.sun.com> (May 1995).
- [GUIL91] Guillemont, M., Lipkis, J., Orr, D., Rozier, M., "A Second-Generation Micro-Kernel Based UNIX; Lessons in Performance and Compatibility," *1991 Winter USENIX Conference*, pp. 13–21, Dallas, TX (January 1991).
- [HOLZE94] Hölze, U., Ungar, D., "Optimizing Dynamically-Dispatched Calls with Run-Time Type Feedback", *Proceedings of the 1994 SIGPLAN Conference on Programming Language Design and Implementation*, Orlando, FL (June 1994).
- [ILLU94] "Illustra DataBlade Developer's Kit Architecture Manual, Release 1.1," Illustra Information Technologies, Oakland, CA (1994).
- [LEE94] Lee, C.-H., Chen, M., Chang, R. C., "HiPEC: High Performance External Virtual Memory Caching," *Proceedings of the First Usenix Symposium on Operating System Design and Implementation*, pp. 153–164, Monterey, CA (November 1994).
- [LISK95] Liskov, B., Curtis, D., Day, M., Ghemaway, S., Gruber, R., Johnson, P., Myers, A., "Theta Reference Manual," MIT LCS Programming Methodology Group Memo 88 (February 1995).

[MAY87] May, C., "MIMIC: A Fast System/370 Simulator," *Proceedings of the SIGPLAN '87 Symposium on Interpreters and Interpretive Techniques*, in *SIGPLAN Notices*, 22, 7, pp. 1-13, St. Paul, MN (July 1987).

[MCCAN93] McCanne, S., Jacobson, V., "The BSD Packet Filter: A New Architecture for User-level Packet Capture," *1993 Winter USENIX Conference*, San Diego, CA (January 1993).

[MCVOY96] McVoy, L., Staelin, C., "lmbench: Portable Tools for Performance Analysis," *1996 USENIX Conference*, San Diego, CA (January 1994).

[MOGUL87] Mogul, J., Rashid, R., Accetta, M., "The Packet Filter: An Efficient Mechanism for User-level Network Code," *Proceedings of the 11th SOSP*, pp. 39-52, Austin, TX (November 1987).

[NELS91] *Systems Programming with Modula-3*, Nelson, G., ed., Prentice Hall, Englewood Cliffs, NJ (1991).

[RFC1321] Rivest, R., "The MD5 Message-Digest Algorithm," *Network Working Group RFC 1321* (April 1992).

[RITCH84] Ritchie, D., "A Stream Input-Output System," *AT&T Bell Laboratories Technical Journal*, 63, 8 pp. 1897-1910 (October 1984).

[ROSE91] Rosenblum, M., Ousterhout, J., "The Design and Implementation of a Log-Structured File System," *Proceedings of the 13th SOSP*, pp. 1-15, Pacific Grove, CA (October 1991).

[TPCB90] Transaction Processing Performance Council, "TPC Benchmark B," Standard Specification, Waterside Associates, Fremont, CA (1990).

[THEK94] Thekkath, C., Levy, H., "Hardware and Software Support for Efficient Exception Handling," *Proceedings of ASPLOS VI*, pp. 110-119, San Jose, CA (October 1994).

[WAHBE93] Wahbe, R., Lucco, S., Anderson, T., Graham, S., "Efficient Software-Based Fault Isolation," *Proceedings of the 14th SOSP*, pp. 203-216 Asheville, NC (December 1993).

[YUHURA94] Yuhara, M., Bershad, B., Maeda, C., Moss, J., "Efficient Packet Demultiplexing for Multiple Endpoints and Large Messages," *1994 Winter USENIX Conference*, pp. 153-166, San Francisco, CA (January 1994).

Christopher A. Small (chris@eecs.harvard.edu) is a Ph.D. candidate in Computer Science at Harvard University. His research interests include the interaction of language systems, database systems, and operating systems. He received his B.A. in Mathematics and his M.A. in Computer Science from Boston University in 1984, where he was a Trustee Scholar. He worked at an unreasonably large number of companies in the Boston area before entering Harvard's Ph. D. program in 1993.

Margo I. Seltzer (margo@eecs.harvard.edu) is an Assistant Professor of Computer Science in the Division of Applied Sciences at Harvard University. Her research interests include file systems, databases, and transaction processing systems. She is the co-author of several widely-used software packages including database and transaction libraries and the 4.4BSD log-structured file system. Dr. Seltzer spent several years working at start-up companies designing and implementing file systems and transaction processing software and designing microprocessors. She is a Sloan Foundation Fellow in Computer Science and was the recipient of the University of California Microelectronics Scholarship, the Radcliffe College Elizabeth Cary Agassiz Scholarship, and the John Harvard Scholarship. Dr. Seltzer received an A.B. degree in Applied Mathematics from Harvard/Radcliffe College in 1983 and a Ph. D. in Computer Science from the University of California, Berkeley in 1992.

Alpha and AlphaStation are trademarks of Digital Equipment Corporation. HotJava, Java, SPARCstation, and Solaris are trademarks of Sun Microsystems. Omniware and omniC++ are trademarks of Colusa Software. DataBlade is a trademark of Illustra Information Technologies. HP-UX and PA-RISC are trademarks of Hewlett-Packard. UnixWare is a trademark of Novell. UNIX is a trademark of X/Open.

An Extensible Protocol Architecture for Application-Specific Networking

Marc E. Fiuczynski
Brian N. Bershad
`{mef,bershad}@cs.washington.edu`

Department of Computer Science and Engineering
University of Washington
Seattle, WA 98195

Abstract

Plexus is a networking architecture that allows applications to achieve high performance with customized protocols. *Application-specific* protocols are written in a typesafe language and installed dynamically into the operating system kernel. Because these protocols execute within the kernel, they can access the network interface and other operating system services with low overhead. Protocols implemented with *Plexus* outperform equivalent protocols implemented on conventional monolithic systems. *Plexus* runs in the context of the *SPIN* extensible operating system.

1 Introduction

This paper describes the design and implementation of *Plexus*, a protocol architecture that allows arbitrary applications to define application-specific protocols. *Plexus* allows protocol processing to be tailored using application-level knowledge, thus providing the framework for supporting new protocols [CSZ92], and implementing optimizations to existing protocols such as integrated layer processing and application level framing and buffering strategies[CT90].

A key aspect of *Plexus* is that application protocol code executes within a kernel-level protocol graph that can be dynamically changed as applications come and go. Once in the kernel, *protocol extension code* can access physical devices and operating system services, such as virtual memory and scheduling, with low overhead, enabling protocols to execute with high efficiency. For the same reason, protocol processing consumes fewer CPU cycles when compared to conventional monolithic implementations.

This research was sponsored by the Advanced Research Projects Agency and by an equipment grant from Digital Equipment Corporation. Fiuczynski was partially supported by a National Science Foundation GEE Fellowship. Bershad was partially supported by a National Science Foundation Presidential Faculty Fellowship.

Although application-specific protocol code executes in the kernel, *Plexus* ensures that it does not compromise overall system safety. We rely on two strategies to ensure safety with exceptionally low overhead. First, application protocol code running in the kernel is written in a typesafe programming language which guarantees that the code respects the interface boundaries against which it was compiled. Second, using link-time control services, *Plexus* restricts direct access to lower level interfaces, ensuring that applications do not snoop or spoof network packets. The strategies are functionally identical to, although less costly than, those found in conventional operating systems where application-specific protocols execute in user-space [TNML93, MB93]; the first prevents arbitrary kernel memory from being accessed, while the second prevents arbitrary kernel functions from being called.

We have built *Plexus* in the context of the *SPIN* extensible operating system [BSP⁺95]. The system runs on DEC ALPHA workstations, and supports a range of protocols, including IP, ARP, ICMP, UDP, TCP and HTTP. Between a pair of ALPHAs running *Plexus*, we have measured an application-to-application round trip packet latency with UDP of less than 600 μ secs on an Ethernet, 350 μ secs on a Fore ATM network, and 300 μ secs on a DEC T3 network. In contrast, the same protocols implemented in *DIGITAL UNIX* using the same device drivers are substantially slower despite the fact that the DEC implementation has been highly optimized [CFF⁺93].

1.1 Motivation

A protocol that performs well for one class of applications can be a bottleneck for others. For example, applications that perform large bulk data transfers over wide area networks are best served by a protocol implementation that provides large local buffers. On the other hand, a connection-oriented protocol that is used for many small transactions is best served by an implementation that minimizes connection lifetime. An application might also benefit from a protocol that is

specific to the application itself, rather than just an implementation of an existing protocol. For example, applications where data integrity is optional such as audio and some flavors of video might use an implementation of UDP for which the checksum has been disabled. This application-specific approach violates the strict definition of the protocol, but, when agreed upon by the communicating applications, is a legitimate way to improve performance.

The protocol architecture in conventional operating systems does not easily accommodate application-specific protocols. First, application-specific code often runs substantially slower than native kernel protocol code [BFM⁺94, RH91, Bir93, vRHB94]. Second, the failure of a protocol module can cause the entire operating system to fail [SMP92, Tho95]. In response to this, some systems only allow the “superuser” to define new protocols [HPAO89, Sun, Wel95], greatly limiting the set of applications for which a new protocol can be defined. In other systems, the protocols must be defined when the system is built [OP92]. Finally, the overall structure of protocols in most commercial systems does not encourage small localized changes.

Plexus offers application developers a protocol architecture having five properties key to the construction of application-specific solutions:

- *Performance*. The system allows an application-specific solution to perform better than the general solution provided by the operating system vendor. Protocol extensions run in the kernel’s address space. This places the protocol close to the network device, eliminates the need to copy data to user space, simplifies process scheduling, and enables resource management decisions using application-level knowledge.
- *Safety*. The use of an application-specific protocol does not compromise the safety of other applications or the operating system. Extensions are isolated from the system and one another through the use of a typesafe language and inexpensive access control mechanisms.
- *Openness*. An application, regardless of its privilege level, may define application-specific protocols, with only marginal performance advantages available to users of higher privilege. This property comes from the inexpensive access control and indirection mechanisms used by *Plexus*.
- *Runtime adaptation*. Applications may add extensions to kernel at any point during the system’s execution without requiring superuser privileges or a system reboot. *Plexus* allows extensions to be safely loaded and unloaded into a running system, so that they can come and go with their corresponding applications.
- *Incremental adaptation*. The level of effort required to make a change to an existing protocol graph is roughly proportional to the size of

the change. The *Plexus* protocol architecture has a modular structure with well-defined interfaces between components. It is defined as a protocol graph with fine-grained nodes implementing portions of protocol functionality.

The rest of this paper describes *Plexus* and is organized as follows. In Section 2 we discuss *SPIN*, the operating system in which we have implemented *Plexus*. In Section 3 we describe the design and implementation of *Plexus*. In Section 4 we present some microbenchmarks that demonstrate the system’s performance. In Section 5 we describe the use and performance of the system in building several application-specific protocols. In Section 6 we survey related work. Finally, in Section 7 we conclude.

2 Overview of the SPIN operating system

SPIN is an operating system that can be dynamically and safely specialized to meet the performance and functionality requirements of applications [BSP⁺95]. Applications define system extensions in Modula-3 [Nel91], which is an Algol-like typesafe programming language. Extensions are dynamically linked into the kernel virtual address space, where they can access other operating system services with low latency. Low latency is important because it enables *fine-grained* interaction between the application and the operating system.

The *SPIN* kernel is written in Modula-3 with the exception of the device drivers which are written in C and borrowed from the *DIGITAL UNIX* source tree. The kernel itself implements threads, virtual memory, and device management. The virtual memory service is used to implement address spaces, which allows applications to execute in their own address space and be written in any language.

In addition to conventional kernel services, *SPIN* also provides *extension* services that allow application-specific services to be integrated into a running system. The extension services address two integration problems: how to safely install code into the kernel’s address space, and how to safely attach that new code to existing kernel services.

The “install” problem is solved by *SPIN*’s dynamic linker [SFPB96], which accepts extensions implemented as partially resolved object files that have been signed by our Modula-3 compiler. The linker resolves any unresolved symbols in the extension against the *logical protection domain* against which the extension is being linked. A logical protection domain defines a set of visible interfaces which provide access to procedures and variables. For example, there is one logical protection domain that includes all interfaces within the kernel (few extensions have access to this domain). There is also a kernel domain that contains

the interface for allocating packet buffers (most extensions have access to this domain). If an extension references a symbol that is not contained within the logical protection domain against which it is being linked, the link will fail and the extension will be rejected. Logical protection domains are first-class kernel resources; they are referenced by typesafe pointers (capabilities), and can be created, copied, and passed around. In this way, different extensions can be given access to different services.

The “attach” problem is addressed by *SPIN*’s dynamic event dispatcher, which communicates *events* to *event handlers*. An *event* is raised by a kernel service or extension code to announce a change in system state or to request a service. For example, the network device drivers raise the event `Ethernet.PacketRecv` to indicate the arrival of a new ethernet packet.

Events are defined and raised using the syntax of procedure declaration and call. That is, the event `Ethernet.PacketRecv` is declared as some procedure `PacketRecv` within an interface `Ethernet` as:

```
INTERFACE Ethernet;
IMPORT Mbuf;
PROCEDURE PacketRecv(READONLY m: Mbuf.T);
...
END Ethernet;
```

The event is raised by “calling” the procedure as in `Ethernet.PacketRecv()`.

Applications interested in the occurrence of an event register an *event handler* with the *SPIN* dispatcher. A handler is a procedure that is executed in response to a specific event. Any extension that can name a particular event (that is, which is linked against the logical protection domain in which the event is defined) may raise the event. More than one handler may be installed on an event, and the overhead of invoking each handler is roughly one procedure call.

An event handler can be associated with a *guard*, which defines an arbitrary predicate that is evaluated by the dispatcher prior to the handler being invoked. If the predicate is true when the event is raised, then the handler is invoked, otherwise the handler is ignored. Guards permit extensions and the base system to separate the specification of what should happen from when it should happen. Extensions define the operations that occur in response to events, whereas guards ensure that those operations happen only at the proper time. *Plexus* relies on guards to implement *packet filters* [MRA87] that correctly route packets through the protocol graph to particular implementations of a protocol.

3 The *Plexus* Architecture

Plexus allows applications to define new protocols or to change the implementation of existing protocols. The system is structured as a protocol graph of event raisers and event handlers. Nodes in the graph represent

protocols for which protocol events are raised to announce or request the passage of a packet through the node. The protocol processing functions are implemented by event handlers that are “attached” to the protocol event name.

Two fundamental events defined by all protocols are `PacketSend` and `PacketRecv`. These events roughly correspond to the `output` and `input` functions found in BSD style protocol stacks, and serve as the binding points for extending a protocol with customized implementations. Access to these events is controlled by a protocol-specific manager, which ensures that applications neither spoof nor snoop packets. Packets sent by an application are pushed down the graph through each protocol’s `PacketSend` event until they reach the actual device. Packets received from the network are pushed up through the protocol graph according to each protocol’s `PacketRecv` event.

Figure 1 shows the graph structure for a TCP and UDP stack. The structure of the graph is a decision tree, with the network device and application extensions forming end-points in the graph. Edges indicate the flow of packets through the protocol stack. Each level of the graph defines a guard and event handler procedure. The guard procedure acts as a packet filter, limiting packets whose headers are not matched by the guard’s predicate on either input (to prevent snooping) or output (to prevent spoofing). The protocol processing functions are implemented by the event handlers that are attached to the protocol event name. Packets sent by the application are pushed down the graph until they reach the actual device. Packets received from the network are pushed up through the protocol graph by the events raised in the preceding protocol layer.

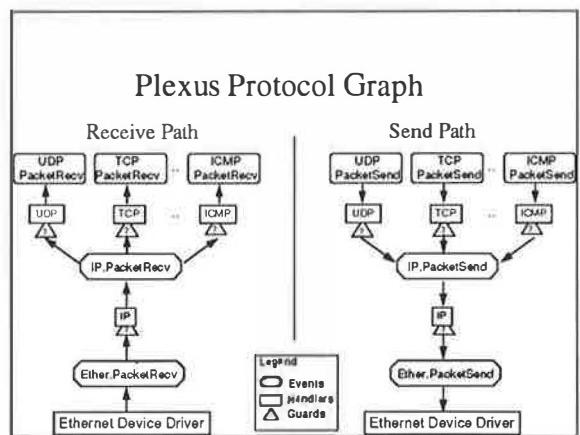


Figure 1: This figure shows the basic structure of protocol stacks under *Plexus*. Each guard is responsible for demultiplexing through one layer of the stack. Each handler is responsible for pushing the packet up to the next layer. A guard (indicated by a “?”) returns true if fields in the header satisfy its predicate.

3.1 Controlling access to protocols

Protection reflects the canonical “mechanism vs. policy” issue for any system. As mentioned, *SPIN* and *Plexus* rely on a typesafe language and controlled linking as the protection mechanism. In this subsection, we discuss the protection policy.

There are two dimensions to protocol protection in *Plexus*: spoofing (sending a packet with an illegitimate source field) and snooping (illegitimately receiving a packet). Both are prevented through the use of *protocol managers* which ensure that a packet is never delivered to, nor accepted from, an illegitimate protocol graph node. It is the responsibility of the protocol manager to define the notion of “legitimacy” with respect to a protocol.

The manager provides a protocol specific interface for accessing and extending the protocol functions. It installs event handlers and guards on the behalf of untrusted applications. Consequently, application-specific extensions do not ever directly install their handlers on protocol events. However, once the handler has been installed, the dispatcher will route control directly to the handler (without going through the intermediate protocol manager). This model of protection is similar to the one used by the Mach user-level protocol library, where the UNIX server installs packet filters on behalf of the library’s requests [MB93].

To send a packet, an untrusted, higher-level protocol must obtain the right to “raise” the *PacketSend* event that transfers the packet to a lower level protocol. Protocol managers prevent spoofing at this level by defining a *PacketSend* event specific to a legitimate sending endpoint. The handler associated with that event can either verify that the contents of the outgoing packet’s source field match the endpoint, or more simply overwrite the source field. The former is useful for debugging protocols, while the latter provides the best performance.

Multiple protocol implementations

Plexus supports multiple implementations of the same protocol for different endpoints in the same way that it supports multiple protocols – different handlers are fired in response to different guard predicates evaluating true. For example, suppose that there are two TCP implementations, *TCP-standard* and *TCP-special*. Both handle the event *IP.PacketRecv*, but the first uses a guard which processes all TCP packets but those destined for the second, while the second handles only those packets destined for the particular set of ports for which it is responsible.

3.2 Safely and efficiently handling packets

Guards and handlers inspect a packet’s payload as it traverses through the protocol graph. Since guards and handlers are written in Modula-3, and Modula-3

is a strongly typed language, the payload must have a type. More importantly, nodes in the graph must be able to cast packets from less specific types (for example, an array of bytes in a device buffer) to more specific types (for example, an Ethernet header followed by an IP header followed by a TCP header). Casting such as this is commonplace in networking software, but Modula-3 as defined lacks sufficient support to cast the primary form of an incoming packet – an array of bytes – safely into a Modula-3 type. The safe alternative, copying, imposes unacceptable overhead. Although unsafe casts are possible using Modula-3’s *LOPHOLE* operator, there is no way to ensure that the operator is used in only safe ways.

To allow for safe casting operations, we have defined a new operator for Modula-3 that converts an array of bytes to a restricted Modula-3 type [HFC⁺96]. The signature of the new operator is *VIEW(a,T):T*. The result of *VIEW(a,T)* is the expression *a*’s bit pattern interpreted as a value of type *T*, which must be a scalar type or an aggregate of scalar types.

There are two major benefits that we gain as a result of the *VIEW* operator. First, it enables Modula-3 programs to safely view external data structures as instances of Modula-3 types without copying. Second, the array of bytes can be efficiently accessed with a more expressive Modula-3 type.

The code fragment in Figure 2 illustrates the use of *VIEW* in accessing packets from within a guard. It also shows how a guard is installed on an event using the Ethernet protocol manager.

3.3 Rapidly processing packets in response to interrupts

Protocols which require little processing for each incoming packet exhibit the best performance when they can run at interrupt level. For example, active messages [vECGS92] require a protocol that does little more than reference memory and reply with an acknowledgement. If such protocols run in a separate thread, outside of an interrupt context, then they will have unnecessarily large latency.

Only privileged device driver code – the bottom of the *Plexus* protocol graph – runs directly in response to network device interrupts. That code, though, may delegate some of its processing to extensions. The critical requirement of an extension so delegated is that it (a) return quickly, and (b) not block. The first requirement reduces the likelihood that an interrupt is lost during the handling of a previous one. The second simplifies the delivery of interrupts, enabling them to be processed in the context of “special” lightweight kernel threads.

We have introduced a small amount of compile-time support [HFC⁺96] to enable a delegating protocol manager, such as the Ethernet layer, to ascertain whether a potential event handler is a “good” candidate for running within an interrupt context. We

```

MODULE ActiveMessages;
IMPORT Mbuf, Ethernet;
PROCEDURE Guard(READONLY m:Mbuf.T):BOOLEAN =
BEGIN
  (* View Ethernet header using Modula-3 type. *)
  (* WITH creates an alias to VIEW'ed header. *)
  WITH etherHdr = VIEW(m.payload[0],Ethernet.T) DO
    RETURN etherHdr.type = ActiveMessageProtoNum;
  END;
END Guard;

PROCEDURE Handler(READONLY m:Mbuf.T)=
BEGIN
  ... (* Do active message handling. *)
END Handler;

BEGIN
  (* Install on Ethernet event *)
  Ethernet.InstallHandler(Ethernet.PacketRecv,
    ActiveMessageGuard, ActiveMessageGuard);
END ActiveMessages.

```

Figure 2: The active message guard/handler pair is installed on the `Ethernet.PacketRecv` event. The guard is invoked for each incoming packet, and acts as a packet filter discriminating on the Ethernet type field. It uses the `VIEW` operator to safely cast a Modula-3 type on the payload array. The code relies on Modula-3's `WITH` statement, which creates an alias of the `VIEW`'ed array, to avoid unnecessary copying.

define a good candidate as one that *can* be asynchronously terminated without damaging important state, but leave it to the candidate handler to distinguish itself as such. A procedure for which the implementation can tolerate premature termination without violating any data structure invariants is explicitly labeled as **EPHEMERAL**. An obvious restriction on ephemeral procedures is that they only call other ephemeral procedures. Our compiler enforces this restriction. Figure 3 illustrates some legal and illegal ephemeral handlers.

A protocol manager can verify that a potential event handler being installed on its `PacketRecv` event is in fact ephemeral by querying the type of the handler procedure. If the procedure is not ephemeral, the manager can reject the handler. Otherwise, the manager can direct the dispatcher to install the handler on the protocol's `PacketRecv` event, and optionally assign a time limit which, if exceeded during handling, will cause the handler to be prematurely terminated. For example, we have extended the protocol graph in Figure 1 to support active messages [vECGS92] over Ethernet. To minimize latency, the active message handlers execute in the network interrupt handler. On the receive path, the extension defines a guard that distinguishes active messages from other incoming Ethernet packets, and provides an ephemeral event handler to process the active message packet. When an active message packet arrives from the Ethernet, the guard will dispatch on the Ethernet type field, and the *SPIN* dispatcher will invoke the corresponding event handler. If the active

```

EPHEMERAL
PROCEDURE Enqueue(q:Queue, READONLY m:Mbuf.T)=
BEGIN
  NonBlockingQueue.Enqueue(q, p);
END Enqueue;

EPHEMERAL
PROCEDURE GoodHandler(READONLY m: Mbuf.T) =
BEGIN
  (* Enqueue incoming packet *)
  Enqueue(ipQueue, m);
END GoodHandler;

(* This procedure is not declared as *)
(* EPHEMERAL *)
PROCEDURE NotEphemeral(READONLY m: Mbuf.T) =
BEGIN ... END NotEphemeral;

EPHEMERAL
PROCEDURE IllegalHandler(READONLY m: Mbuf.T) =
BEGIN
  (* This procedure won't compile *)
  (* because NotEphemeral is not ephemeral *)
  NotEphemeral(m);
END IllegalHandler;

```

Figure 3: Ephemeral and non-ephemeral handlers. A manager for a protocol that runs directly in response to a device interrupt would allow the installation of `GoodHandler` on its `PacketRecv` event, but should not allow the installation of `NotEphemeral`. The compiler will generate an error when compiling `IllegalHandler` since it calls a procedure that is not ephemeral.

message handler exceeds its time allotment, it will be terminated.

3.4 Sharing packet buffers

A packet can be shared across multiple levels of the protocol graph on both the send and receive paths. Although multiple extensions can view the network data, they cannot modify it without making a copy of it first. We achieve this effect by passing packets through the protocol graph as *read-only* buffers.¹ Figure 4 provides an example of read-only buffers using Modula-3's `READONLY` type specifier. The `BadPacketRecv` procedure overwrites the contents of the packet. However, the code fragment will be rejected by the compiler as the left-hand side of the statement is a read-only variable. An extension such as `GoodPacketRecv` must allocate a new buffer and copy the read-only contents in order to modify it (i.e., an explicit copy-on-write).

¹ *Plexus* uses *nibus* (the Berkeley memory buffer implementation) to pass packets through the protocol graph. A primary advantage of *nibus* is that they are directly used by most UNIX device drivers.

```

TYPE Mbuf.T = RECORD
  m_hdr : mh_hdrT;
  m_data : ARRAY [1..MLEN] OF Bytes;
END;

PROCEDURE GoodPacketRecv(READONLY m: Mbuf.T) =
  VAR p: Mbuf.T;
  BEGIN
    p := m;
    FOR i := FIRST(p.m_data) TO LAST(p.m_data) DO
      (* overwrite packet data *)
      (* will be allowed by compiler. *)
      p.m_data[i] := 0;
    END;
  END GoodPacketRecv;

PROCEDURE BadPacketRecv(READONLY m: Mbuf.T) =
  BEGIN
    FOR i := FIRST(m.m_data) TO LAST(m.m_data) DO
      (* overwrite packet data *)
      (* will be rejected by compiler *)
      m.m_data[i] := 0;
    END;
  END BadPacketRecv;

```

Figure 4: A pair of packet receive handlers. One is legal because it does not modify its argument. The other is not legal and will be rejected by the compiler.

3.5 Summary

Plexus defines a protocol graph in which applications can introduce new nodes (handlers) and edges (guards) at runtime. Safety is ensured by restricting an extension's access to underlying interfaces, and by filtering packets before they arrive at an extension's handler.

Although we have implemented *Plexus* in the context of *SPIN*, we believe that the general structure of our protocol system, in particular, the graph architecture, the interfaces, and the protocol protection model, could be implemented in more conventional systems such as UNIX provided they support kernel extensions in a safe fashion. Fundamentally, our protocol architecture requires that the kernel export two facilities: dynamic linking/unlinking and in-kernel firewalls. Dynamic linking/unlinking is necessary to install and remove new protocol extensions into and from the kernel. An in-kernel firewall makes it possible to run user code within the kernel without compromising system integrity. Typesafe languages are not the only mechanism for implementing firewalls. Alternative strategies include interpreted languages [GM, Ous94] and software based fault isolation techniques [WLAG93].

4 Performance

In this section we describe basic latency and throughput measurements for *Plexus* on a range of networking devices. Specifically, we examine the system's performance for UDP and TCP, and compare it to the same protocols running on *DIGITAL UNIX*, a commercial operating system.

All measurements were done using a version of the *SPIN* kernel from November 1995. We used the DEC SRC Modula-3 compiler (release 3.5.2) to compile the bulk of the kernel and its extensions. We used the DEC C compiler (from *DIGITAL UNIX* 3.2) to compile the system's device drivers, as well as version 3.2 of the *DIGITAL UNIX* operating system. Both *SPIN* and *DIGITAL UNIX* run on members of the DEC Alpha workstation family. For the measurements in this paper, we used DEC model 3000/400 workstations, which have an Alpha 21064 processor running at 133Mhz. Each workstation was equipped with 64MB of RAM, a 10Mb/sec Ethernet, a 155Mb/sec Fore TCA-100 ATM interface on the TurboChannel I/O bus, an experimental 45Mb/sec Digital T3 network adapter, and an SFB framebuffer. All Ethernet performance measurements were made between two machines on a private Ethernet segment. Our ATM cards are connected to a ForeRunner switch. The T3 measurements were made by connecting two workstations back-to-back. Our ATM network interface cards use programmed I/O, limiting maximum bandwidth to the rate with which the CPU can read the data from the network adapter. With our hardware configuration, we have been unable to achieve greater than 53Mb/sec when transferring data reliably between two device drivers. The T3 adapter uses DMA, and is able to deliver 45Mb/sec with minimal CPU involvement.

4.1 Protocol latency

The round-trip latency of a protocol reflects the overhead induced by the protocol as it transfers bytes from sender to receiver and back to receiver. Figures 5 shows the round-trip latency for small (8 byte) UDP/IP messages between a pair of application-specific functions on *SPIN/Plexus* and *DIGITAL UNIX* on Ethernet, the Fore ATM interface, and the DEC T3 interfaces. Both *SPIN* and *DIGITAL UNIX* use the same device drivers.

In *DIGITAL UNIX*, the application code executes at user-level, and each packet sent involves a trap and a copy-in as the data moves across the user/kernel boundary. In the worst case, the receive side must schedule the user process, copy the packet to user-space, and context-switch. In *SPIN*, the application code executes as an extension in the kernel, where it has low latency access to both the device and data. Each incoming packet causes a series of events to be generated for each layer in the protocol stacks (Ethernet/ATM, IP, UDP). The figure shows two bars for

Plexus for each device, one labeled **interrupt** and one labeled **thread**. The **interrupt** case reflects the round trip latency when the application-specific handler runs as an **EPHEMERAL** procedure that executes at interrupt level. The **thread** case reflects the overhead when the protocol and application handlers run in separate threads, with each event raise creating a new thread.

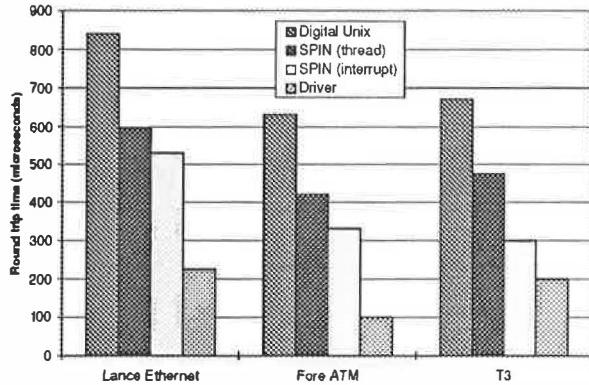


Figure 5: *UDP Round trip network send/receive time for small (8 byte) packets when using different networking hardware with *Plexus* and *DIGITAL UNIX*.*

The latency measurements show that cross-machine communication has lower latency when the target and source processes are able to send directly from the kernel. As mentioned, both systems use the same network device driver. Consequently, the difference between the performance of the two systems reflected in Figure 5 is due strictly to differences in operating system and protocol structure. The figure also shows the minimal round trip time using our hardware as measured between the device drivers. In tests using a faster device driver for *SPIN*, we measured a round-trip UDP latency of 337 μ secs on Ethernet and 241 μ secs on ATM. (We did not write a faster device driver for T3).

4.2 Throughput

Throughput is much less sensitive to operating system and application overheads than latency. Both *Plexus* and *DIGITAL UNIX* use the same TCP/IP implementation and device drivers. The *Plexus* TCP/IP implementation is one of the few cases in *SPIN* where we allow code not written in Modula-3 to be downloaded into the kernel. This code comes from a commercial vendor and we assume it to be safe, as it is conformant to interfaces and contains no illegal loads or stores. Consequently, the measured throughput for *Plexus* and *DIGITAL UNIX* are nearly identical. Specifically, for Ethernet, we saw 8.9 Mb/sec, and for the Fore ATM

card, we saw 27.9 Mb/sec with *DIGITAL UNIX* and 33 Mb/sec with *Plexus*.²

5 Application-specific protocols

In this section we describe two application-specific protocols that we have built using the *Plexus* architecture.

- A *network video protocol* that shows how an application-specific protocol can deliver greater throughput with lower CPU utilization than a general protocol.
- A *packet forwarding protocol* that quickly reroutes TCP and UDP packets on a port-specific basis.

In the first case, we implemented multicast semantics for the UDP protocol. In the second case, we modified the TCP and UDP receive paths to simply reflect packets to second host. In both cases, we compare the performance of these application-specific protocols against standard protocols.

5.1 Network video

We have used our networking architecture to implement a networked video system consisting of a server that multicasts video clips to a set of clients. The server consists of one extension that reads video frame-by-frame off of the disk using *SPIN*'s file system interface. Because the video server extension is co-located with the kernel, it does not have to copy the data across the user/kernel boundary to send the disk block data back out over the network. The advantage of this structure is that video frames leave the server quickly as they travel from disk to network.

The server sends each frame as a UDP packet over the network to a number of clients. A video stream is composed of 30 frames per second. Each client receives one stream from the server. We experimented with 1–30 streams to determine when the server would fail to meet its deadline of sending each client a video stream.

Figure 6 shows the processor utilization on the server as a function of the number of client streams for our video system running over the T3 network. At 15 streams, both *SPIN* and *DIGITAL UNIX* saturate the network, but *SPIN* consumes only half as much of the processor. Compared to *DIGITAL UNIX*, *SPIN* can support more clients on a faster network, or as many clients on a slower processor. We do not present data for the Fore interface, because for both systems, the majority of the server's CPU resources are consumed by the programmed I/O that copies data to the network one word at a time (recall that T3 uses DMA).

²A bug in *SPIN*'s DMA support prevented us from measuring TCP throughput in this one case.

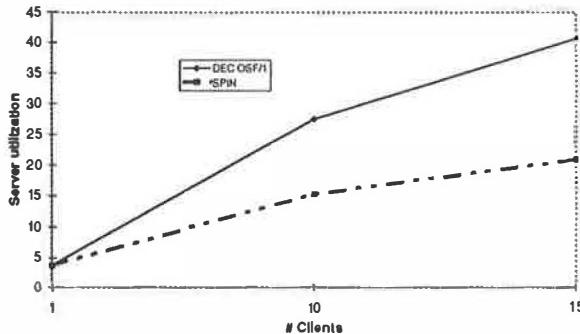


Figure 6: Utilization of the server's CPU as a function as a function of the number of client video streams.

The client

On the client, an extension awaits incoming video packets with the protocol graph shown in Figure 1. The client extension checksums and decompresses the image and displays it directly to the screen's framebuffer. The current implementation makes two passes over the data, one pass for the checksum and another to decompress the image.

The client viewer is a good candidate for the integrated layer processing optimizations suggested by Clark [CT90]. We use the same video display code for both the *SPIN* application extension and the *DIGITAL UNIX* application. To achieve this, the video client implemented on *DIGITAL UNIX* maps the framebuffer directly into its address space and uses the same viewer code as the *SPIN* implementation. We expected that the overhead incurred for the data and control transfer to be significantly higher for *DIGITAL UNIX* compared to *SPIN*. However, the CPU utilization between the two operating systems was similar.

The similarity in performance is due to the high overhead in writing video data to the framebuffer. Thus, the performance of the video client is limited by the write bandwidth of the framebuffer hardware rather than overhead incurred by the operating system. Writing to the framebuffer memory is a factor of 10 times slower than writing to standard RAM. Displaying the video data is a factor of up to 50 times slower than all of the low-level OS operations combined. Clearly, the benefits of application specific networking protocols are most notable when the protocol processing is not dominated by application processing. In particular, customized protocols are most appropriate for applications that require low latency or high throughput with lower CPU utilization. In the video client, the benefits of a customized video protocol are offset by the fact that the application spends a signifi-

cant amount of time (more than 90%) writing data to the framebuffer, rather than processing video packets from the network. We expect that with better video hardware, such as the DEC J300 device [PP93], the dominant performance bottleneck will be the protocol processing rather than the application processing.

Protocol forwarding

Plexus can be used to provide protocol functionality not generally available in conventional systems. For example, we have built a forwarding protocol that can be used to load balance service requests across multiple servers. To do this, an application installs a node into the *Plexus* protocol graph that redirects all data and control packets destined for a particular port number to a secondary host. We have implemented a similar service using *DIGITAL UNIX* with a user-level process that splices together an incoming and outgoing socket. The *DIGITAL UNIX* forwarder is not able to forward protocol control packets because it executes above the transport layer. As a result it cannot maintain a protocol's end-to-end semantics. In the case of TCP, end-to-end connection establishment and termination semantics are violated. A user-level intermediary also interferes with the protocol's algorithms for window size negotiation, slow start, failure detection, and congestion control, possibly degrading the overall performance of connections between the hosts. Moreover, on the user-level forwarder, each packet makes two trips through the protocol stack where it is twice copied across the user/kernel boundary. Figure 7 shows the impact that this additional work has on TCP forwarding performance.

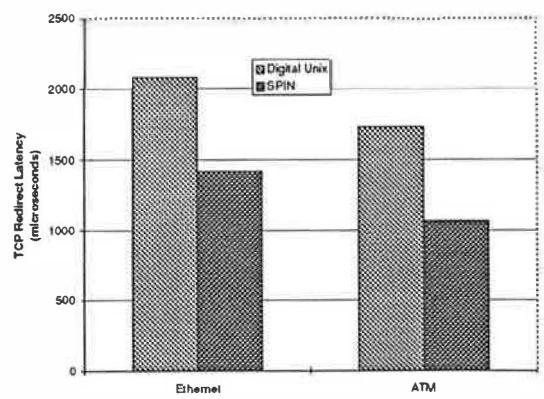


Figure 7: TCP redirection latency using *Plexus* and *DIGITAL UNIX*. The *DIGITAL UNIX* implementation runs at user-level and is unable to respect end-to-end TCP semantics.

6 Related Work

Many operating systems provide an interface that enable code to be installed into the kernel at runtime. Examples include dynamically linked device drivers, system calls, and networking protocols [Sun, IBM93, Wel95]. However, the extensions in these systems can see all kernel symbol names, giving them free-reign over the internals of the system. For example, a dynamically linked extension has complete access to the virtual memory subsystem, thus can manipulate data belonging to all currently running applications. In these systems, application-specific extensibility is not safely possible, as any system extension can bring down the entire system or violate system integrity. Fall and Pasquale describe a *splicing* mechanism that enables applications to route data between two devices directly within the kernel [FP93a, FP93b]. Splicing is only a partial solution, however, if applications must manipulate the data, for example, to decrypt, before passing it on to the next stage in the splice.

Several projects have defined protocol structures allow applications to use their own protocols in a safe manner within their address space [TNML93, MB93]. Our architecture resembles their organization by separating protocol send and receive code from end-point management, but allows applications to link code into the kernel virtual address space where they have low latency access to operating system services, buffer management, and the network interface.

The *x*-kernel [OP92] provides a framework for implementing network protocols. In the *x*-kernel, many small protocols, called micro- and virtual- protocols, implement simple processing functions tied together by a complex graph to implement a standard protocol. Similarly, our architecture decomposes the stack into a protocol graph using *SPIN* events and guards. Our protocol event handlers resemble the *x*-kernel's *micro-protocols*. Similarly, our guards are functionally equivalent to *virtual protocols*. The main difference between the two architectures is that we permit users to safely and dynamically extend the protocol graph in the kernel by allowing application-specific protocol code to be placed within the stack.

The Fox project [BHL93] uses a high level language, ML, to implement protocol stacks. Through the use of strongly typed interfaces they can compose arbitrary protocol and experiment with customized protocols. However, their system runs only in user space, making it difficult to achieve good performance with a customized protocol. Moreover, current ML compilers focus on optimizing higher-order functions and polymorphism instead of loops and data representations, with the result being that typical systems code runs 3 to 5 times slower than comparable C code. In contrast, Modula-3 compilers can generate code with quality comparable to widely available C compilers [SSPB96].

Horus [vRHB94] identifies the need for fast and flexible protocols in the context of group communication.

Horus provides a layered architecture for applications to compose customized protocols from a set of predefined basic protocols. All protocols inherit from a basic group which applications extend with group communication features, such as message ordering or flow control. Their work concentrates on providing application specific communication protocols in the context of group communication, whereas our work allows applications to dynamically extend the system's protocol stack.

7 Conclusion

This paper described an application specific networking architecture that allows an application's extension to be co-located with the kernel, thus enabling it to directly access system resources with low overhead. The extension model allows untrusted application extensions to handle system events efficiently inside the kernel. A demonstration of the protocol stack as it services HTTP requests can be found at <http://www-spin.cs.washington.edu>. The source for the protocol architecture can be found on the *SPIN* home page at that site.

Acknowledgements

We would like to thank the members of the *SPIN* group at the University of Washington for their help in building the operating system that hosts *Plexus*. David Becker did the initial port of TCP/IP to *SPIN* that became the basis for the *Plexus* implementation. Emin Gün Sirer and Przemyslaw Pardyak assisted with the measurements presented in this paper. Stefan Savage provided feedback on earlier versions of this paper. Our friends from Digital Equipment Corporation also deserve special thanks. David Boggs supplied us with our T3 interfaces. Finally, Bill Kalsow and others at DEC SRC have made available a high quality implementation of Modula-3, enabling us to concentrate on the high level interface issues that arise in the design of an extensible protocol system.

References

- [BFM⁺94] A. Banerjea, D. Ferrari, B. Mah, M. Moran, D. Verma, , and H. Zhang. The Tenet Real-Time Protocol Suite: Design, Implementation, and Experiences. Technical Report 94-059, International Computer Science Institute, Berkeley, November 1994.
- [BHL93] Edoardo Biagioni, Robert Harper, and Peter Lee. Standard ML Signatures for a Protocol Stack. Technical Report CMU-CS-F~~0~~X-93-01, Carnegie Mellon University, 1993.
- [Bir93] K.P. Birman. The Process Group Approach to Reliable Distributed Computing. *Communications of the ACM*, 36(12), December 1993.
- [BSP⁺95] Brian N. Bershad, Stefan Savage, Przemyslaw Pardyak, Emin Gün Sirer, Marc E. Fiuczynski,

- David Becker, Susan Eggers, and Craig Chambers. Extensibility, Safety and Performance in the SPIN Operating System. In *Proceedings of the Fifteenth ACM Symposium on Operating Systems Principles*, December 1995.
- [CFF⁺93] Chuan-Han Chang, Richard Flower, John Forecast, Heather Gray, William R. Haw, K. K. Ramakrishnan, Ashok P. Nadkarni, Uttam N. Shikarpur, and Kathleen M. Wilde. High-performance TCP/IP and UDP/IP Networking in DEC OSF/1 for Alpha AXP. *Digital Technical Journal*, 5(1), 1993.
- [CSZ92] David D. Clark, Scott Shenker, and Lixia Zhang. Supporting Real-Time Applications in an Integrated Services Packet Network: Architecture and Mechanism. In *SIGCOMM '92 Conference Proceedings*, pages 14–26, August 1992.
- [CT90] D. D. Clark and D. L. Tennenhouse. Architectural Considerations for a New Generation of Protocols. In *Proceedings of the ACM SIGCOMM '90 Symposium on Communications Architectures and Protocols*, pages 200–208, September 1990.
- [FP93a] Kevin Fall and Joseph Pasquale. Exploiting In-kernel Data Paths to Improve I/O Throughput and CPU Availability. In *Proceedings of the 1993 Winter USENIX Conference*, pages 327–333, January 1993.
- [FP93b] Kevin Fall and Joseph Pasquale. Improving Continuous-media Playback Performance With In-kernel Data Paths. In *Proceedings of the IEEE International Conference on Multimedia Computing and Systems (ICMCS)*, pages 100–109, June 1993.
- [GM] James Gosling and Henry Gilton. The Java Language Environment: A White Paper. <http://java.sun.com>.
- [HFC⁺96] W.C. Hsieh, M.E. Fiuczynski, C. Garrett, S. Savage, D. Becker, and B.N. Bershad. Language Support for Extensible Systems. Submitted to the First Workshop on Compiler Support for Systems Software, November 1996.
- [HPAO89] Norman C. Hutchinson, Larry Peterson, Mark B. Abbott, and Sean O’Malley. RPC in *x*-kernel: Evaluating New Design Techniques. In *Proceedings of the Twelfth ACM Symposium on Operating Systems Principles*, pages 91–101, December 1989.
- [IBM93] IBM Corporation. *AIX Version 3 for RISC System/6000 - Kernel Extensions and Device Support Programming Concepts*, October 1993. SC23-2207-00.
- [JSS94] K. Jeffay, D.L. Stone, and F.D. Smith. Transport and Display Mechanisms for Multimedia Conferencing Across Packet-Switched Networks. *Computer Networks and ISDN Systems*, 26(10):1281–1304, July 1994.
- [MB93] Chris Maeda and Brian N. Bershad. Protocol Service Decomposition for High-Performance Networking. In *Proceedings of the Fourteenth ACM Symposium on Operating Systems Principles*, pages 244–255, December 1993.
- [MRA87] J. Mogul, R. Rashid, and M. Accetta. The Packet Filter: An Efficient Mechanism for User-level Network Code. In *Proceedings of the Eleventh ACM Symposium on Operating Systems Principles*, pages 39–51, 1987.
- [Nel91] Greg Nelson, editor. *System Programming in Modula-3*. Prentice Hall, 1991.
- [OP92] S. W. O’Malley and L. L. Peterson. A Dynamic Network Architecture. *ACM Transactions on Computer Systems*, 10(2), May 1992.
- [Ous94] John K. Ousterhout. *Tcl and the TK Toolkit*. Addison-Wesley Publishing Company, 1994.
- [PP93] Lawrence G. Palmer and Rick S. Palmer. DECspin: A Networked Desktop Videoconferencing Application. *Digital Technical Journal*, 5(2), 1993.
- [RH91] Franklin Reynolds and Jeffrey Heller. Kernel Support for Network Protocol Servers. In *Proceedings of the Second Usenix Mach Workshop*, November 1991.
- [SFPB96] E.G. Sirer, M. Fiuczynski, P. Pardyak, and B.N. Bershad. Safe Dynamic Linking in an Extensible Operating System. Submitted to the First Workshop on Compiler Support for Systems Software, November 1996.
- [SMP92] Andrew Schulman, David Maxey, and Matt Pietrek. *Undocumented Windows*. Addison-Wesley, 1992.
- [SSPB96] E.G. Sirer, S. Savage, P. Pardyak, and B.N. Bershad. Writing an Operating System in Modula-3. Submitted to the First Workshop on Compiler Support for Systems Software, November 1996.
- [Sun] Sun Microsystems. *Solaris - SunOS 5.9 Writing Device Drivers*.
- [Tho95] Tom Thompson. Copland: The Abstract Mac OS, July 1995.
- [TNML93] Chandramohan A. Thekkath, Thu D. Nguyen, Evelyn Moy, and Edward D. Lazowska. Implementing Network Protocols at User Level. In *Proceedings of the ACM SIGCOMM '93 Symposium on Communications Architectures and Protocols*, September 1993.
- [vECGS92] Thorsten von Eicken, David E. Culler, Seth Copen Goldstein, and Klaus Erik Schauer. Active Messages: A Mechanism for Integrated Communication and Computation. In *Proceedings of the 19th International Symposium on Computer Architecture*, pages 256–266, May 1992.
- [vRHB94] R. van Renesse, Takako M. Hickey, and Kenneth P. Birman. Design and Performance of Horus: A Lightweight Group Communications System. Technical Report 94-1442, Cornell University’s Computer Science Department, August 1994.
- [Wel95] Matt Welsh. Implementing Loadable Kernel Modules For Linux. *Dr. Dobbs Journal*, 20(5), 1995.
- [WLAG93] Robert Wahbe, Steven Lucco, Thomas E. Anderson, and Susan L. Graham. Efficient Software-Based Fault Isolation. In *Proceedings of the 14th ACM Symposium on Operating System Principles*, pages 203–216, December 1993.

Linux Device Driver Emulation in Mach

Shantanu Goel and Dan Duchamp

Computer Science Department
Columbia University

Abstract

We describe the design and performance of code added to the Mach microkernel (Mach 4.0, version UK02p21) that permits one to build a Mach kernel that includes unmodified Linux device drivers. We have written emulation code to support all Linux 1.3.35 network and SCSI drivers for the ISA and PCI I/O buses. Emulation increases latency, but very little. The degree depends on both device and operation, and varies from 2 microseconds for receiving small (60 byte) network packets up to 197 microseconds for writing 16KB to an ISA SCSI device.

1 Introduction

We describe the design and performance of code that permits one to build a Mach microkernel (Utah release 4.0, version UK02p21) that includes the completely unmodified source for device drivers that have been written for Linux (version 1.3.35). Our code, which consists of some changes and additions to Mach as well as run-time emulation of Linux calls, handles all of Linux's block drivers, network drivers, and SCSI host adapters for the ISA or PCI buses—53 drivers in all (block drivers for floppy, IDE hard disk, and SCSI; 30 ISA network devices; 4 PCI network devices; 10 ISA SCSI host adapters; and 5 PCI SCSI host adapters).

The motivation for this work is to improve the usefulness of the Mach microkernel on Intel x86 platforms. We are wedded to Mach because some of the research in our laboratory is dependent on its unique features. Our research also needs to incorporate new peripheral devices on a regular basis. Unhappily, because of its small user base, Mach has always had relatively few device

drivers compared to more popular operating systems. Furthermore, many of these drivers are old and do not accommodate recent generations of I/O chips, either by not running them at all or else by failing to take advantage of advanced new features. In part because of developments in multimedia and wireless networking, new I/O peripherals are being invented at a remarkable rate, and Mach's set of device drivers has been steadily falling further behind the hardware base available in the PC world.

This problem eventually became acute for us. We knew we could not obtain either the time or the information to write all the Mach device drivers we wanted; writing new device drivers is often difficult because of the need to obtain access legally to proprietary hardware specifications and/or software, and because one must have a sound understanding of the hardware in order to write a high quality driver. Accordingly, we wondered whether it would be practical to implement Linux device driver emulation within Mach. Because of its relatively large following, Linux has many more device drivers and the rate at which new drivers are added to it outstrips the rate at which new drivers are added to Mach. Linux has driver call-in and call-out interfaces that are well defined and that change slowly. We thought that if we could emulate these interfaces, then we could tap into the current base of Linux drivers, and—possibly with limited further effort to update the emulation—future Linux drivers as well.

Our goal was to be able to compile completely unmodified Linux device drivers into the Mach kernel. We achieved this goal for network, block, and SCSI devices that attach to either the ISA or PCI bus. This paper reports our design, how certain aspects of both Mach and Linux constrained our design, and how our code performs.

2 Background

This section provides necessary background about how device drivers operate in both Mach 4.0 and Linux 1.3.35.

In Linux, drivers may be either statically or dynamically linked into the kernel. Mach supports only statically linked drivers. Consequently, we emulate only statically linked Linux drivers.

Linux has five classes of devices: network, SCSI, block, sound, and character. At this writing, we have tackled only network, block, and SCSI drivers that attach to the ISA and PCI I/O buses.

Linux and Mach view device access differently. Mach is a microkernel designed with portability in mind. It has a single kernel interface for all device types, and this interface is accessed by messages. Within the kernel, a device access request first passes through machine independent code and then to machine dependent code.

Linux is a monolithic operating system that was originally targeted to only the Intel x86 architecture. Its device interfaces are accessed by other parts of the operating system via procedure call, and there is no attempt at machine independence. Each type of device has its own interface.

Figures 1, 2, and 3 give the Mach device interface and Linux interfaces to network and SCSI devices, respectively.

Linux regards SCSI devices as block devices, so the interface to SCSI is the vnode interface; this explains the large number of unimplemented calls in Figure 3. Of course in both systems a device driver must also have an interface that is accessed from below: one procedure per type of interrupt. In the case of network drivers, there are two types of interrupt: *transmit done* and *packet received*. The SCSI interrupts are *command complete* and *bus reset by device*.

Network drivers have essentially no internal structure. There is one procedure to handle each entry point and a few utility routines. However, in both Mach and Linux SCSI drivers have an internal structure. The top layer is “target specific,” example targets are tape and disk/CD-ROM. The target specific layer knows about the physical structure and constraints of one type of device. This layer translates device-specific operations (e.g., read a disk block) into one or more SCSI commands. The middle layer performs bookkeeping chores like queueing and timeouts. The bottom layer is the “host adapter,” which knows how to send one or more SCSI commands to a specific controller chip and return the result(s). The

```
device_open
device_close

device_read           /* synchronous */
device_write
device_read_inband   /* small variant */
device_write_inband

/* these two replace ioctl */
device_get_status
device_set_status

/* set network packet filter */
device_set_filter

device_map
/*
 * Also, asynchronous two-message
 * (request & reply) versions of
 * each of these calls: device_open,
 * device_read, device_write,
 * device_read_inband, and
 * device_write_inband.
 */
```

Figure 1: Mach Kernel Interface to All Devices

interface to the host adapter is given in Figure 4. This interface is the same in both Mach and Linux.

3 Design

This section presents the details of our design. Section 3.1 discusses the relative power of the two device interfaces; i.e., is it possible to emulate the Mach device interface using Linux drivers? The remaining subsections, 3.2 through 3.6, discuss the specific modules of emulation code that we wrote. The implementation consists of about 2000 lines of C.

3.1 Device Interfaces

At the most abstract level of consideration, emulation raises two issues:

1. Emulating any procedure call or variable reference that a Linux driver might make.
2. Ensuring that the combination of emulation code plus a Linux driver are together able to implement all Mach device entry points.

```

probe
open
close
send_packet
ioctl           /* unimplemented */
get_stats
set_multicast_list /* effectively,
                     this is ioctl;
                     link address and
                     promiscuous mode
                     can be set too */

```

Figure 2: Linux Interface to Network Devices

```

open
release      /* same as close */
read
write
ioctl

/* obscure */
change       /* has media changed?
               e.g., CD-ROM removal */
validate     /* flush and re-read
               disk partition tables */
fsync

/* not implemented by any driver */
lseek
readdir
select
mmap
fasync      /* asynchronous sync */

```

Figure 3: Linux Interface to SCSI Devices

The first concern—how Mach can provide the facilities needed by Linux drivers—is addressed in the sections below. This section shows that Linux drivers can implement the Mach interface as well as Mach drivers implement it. In the next few paragraphs we argue that the Mach kernel interface to devices can be implemented by Linux drivers plus a small amount of simple emulation code.

Figures 2 and 3 show that Linux has obvious analogues for most Mach device entry points: `open`, `close`, `read`, `write`, and

```

detect      /* probe for device */
command    /* synchronous */
queue_command /* asynchronous */
reset

```

Figure 4: Interface to SCSI Driver Host Adapter Layer

`ioctl`. Of course, it is possible that certain Mach `ioctl` arguments are not implemented by Linux drivers. In the case of network devices, Mach’s `ioctl` calls (`device_get_status` and `device_set_status`) map not to `ioctl` but to the non-obvious `set_multicast_list`. With the right arguments, `set_multicast_list` can be used to read or write the link address, multicast addresses, and promiscuous read mode. These are the major actions of Mach’s `ioctl`.

There is no Linux analogue for the Mach `device_set_filter` entry point. This entry point is used to associate a packet filter [3] with a driver. However, in Mach neither this entry point nor actual filtering of packets is implemented by device drivers. Instead, Mach has generic routines for installing and executing packet filters, and all device drivers call the generic installation routine. Once installed, a packet filter is called after the network driver has retrieved the incoming packet and copied it into a Mach message. The filter operates on the message contents. At that point, all device-specific functions have been performed, so packet filtering really takes place above the driver level.

The `device_map` entry point permits devices like frame buffers and disks to map their contents into virtual memory. The Linux analogue is `mmap`. As a practical matter, the only drivers that use these calls in either system are frame buffers, so emulation is presently a moot point. If someday it becomes important for a Linux driver to service Mach `device_map` calls, emulation would be easy because of the presence of a generic Mach routine (`blockio_map`) that converts paging activity of the mapped area into device read/write requests.

3.2 In-Kernel Device Interface

Below Mach’s kernel device interface and above the device drivers is a layer of “machine independent” code. This code unpacks request messages into a generic I/O request structure, maps

or moves pages between the calling process and the kernel, and sends a reply message when the device driver is finished with the operation. Also, during a `device_open` operation, this code looks up the device's name in a list called `dev_name_list`. Device drivers add entries to `dev_name_list`. Such entries include the device's name and pointers to routines in the device driver that implement the calls in the kernel's device interface. The idea is that the machine independent code performs all kernel actions that are not truly device-specific.

After acquiring some experience with the machine independent code, we decided to eliminate it because it interferes with the goal of using device drivers from other operating systems.

For example, Linux names IDE drives `ide0`, `ide1`, and so on, but Mach's UNIX server refers to IDE drives as `hd0`, `hd1`, etc. To use Linux drivers in a Mach microkernel that is running the UNIX server, it would be necessary to update the machine independent layer to translate a `device_open` kernel call with an argument of `hd0` into a call to the Linux driver. More generally, extra effort is needed to convert between several conventions defined by Mach's machine independent layer and the corresponding conventions of Linux drivers.

Since the machine independent layer would have to change anyway, and since we thought the type of translation mentioned above is more properly done by the emulation code associated with the Linux driver, we decided to eliminate the machine independent layer. The result is a new implementation of all Mach device interface calls that recognizes emulation as a possibility. For example, when a `device_open` request is received, the kernel calls the `open` routine of each emulation module. An emulation module that recognizes the device name returns a structure that contains pointers to routines that implement each call in the kernel interface. For calls other than `device_open`, the proper routine pointer in the structure is dereferenced immediately upon entry into the kernel. The emulation module is responsible for all actions needed to service a device interface call, including mapping pages and sending a reply. Three emulation modules currently exist: Linux block, Linux network, and old Mach. The old Mach module is the original Mach device code, hacked a bit to conform to the new way of doing things.

3.3 Initialization

Linux assumes that the clock is running at the time drivers configure, but in Mach this is not true. This creates a problem since Linux drivers use clock interrupts to timeout device probe commands. In particular, the initializing driver (which is the only activity running on the machine at the time) initiates a probe command and then polls the clock variable in a loop until the probed device responds with an interrupt or until the clock variable reaches some timeout value. The clock variable is incremented by the handler for clock interrupts. (Mach drivers use a similar method except that, because no time facility is available, the delay loop runs for an "estimated" amount of time.)

We solved this problem by changing Mach to start the clock earlier, and by writing a special clock interrupt handler that is used only during Linux driver configuration. We have also renamed the clock variable that Linux drivers refer to; Mach calls the variable `elapsed_ticks` while Linux calls it `jiffies`. The special handler increments this variable, and we have changed the standard Mach clock handler to do so also. Mach's clock handler could not be used during driver initialization because it manipulates the "current thread" data structure which doesn't exist at that point. We did not fix Mach device drivers after the fact to use measured time rather than estimated time in their delay loops.

3.4 Memory Usage

Mach and Linux make different use of the kernel's address space in two ways: addressing and memory allocation. Both of these differences impact driver emulation.

3.4.1 Addressing

Both Mach and Linux map the kernel into the upper gigabyte of the 32-bit address space. However, Mach sets kernel segment register values to zero, and is linked so as to generate virtual addresses in the range [0xC0000000 .. 0xFFFFFFFF]. In contrast, Linux sets its segment registers to 0xC0000000, and generates virtual addresses in the range [0x0 .. 0x3FFFFFFF]. Linux does this to ease kernel programming: kernel virtual addresses and physical addresses are identical and interchangeable. Consequently, Linux drivers have no provision for translating between physical addresses and kernel virtual addresses, and opera-

tions that require physical addresses (e.g., DMA) would not work if a Linux driver were simply compiled and linked with Mach.

To resolve this difference, we changed Mach's machine-specific memory management module (`pmap`) and linking instructions so that Mach also generates virtual addresses in the range [0x0 .. 0x3FFFFFFF].

3.4.2 Kernel Memory Allocation

An initialization call to a DMA-capable Linux driver includes two parameters which are the start and end addresses of a segment of contiguous, DMA-able¹ physical memory. The driver uses this memory for DMA buffers and for storing its data structures. In Mach, driver initialization occurs after virtual memory is enabled, so the virtual memory system cannot be avoided when searching for memory for driver initialization. The emulation code searches Mach's free page list, looking for a sequence of contiguous pages that lie below the 16MB boundary. The boundary addresses of this segment are passed to the Linux driver.

To service dynamic requests made by Linux drivers for extra DMA buffers, we implemented a new memory allocator that Linux drivers (only) use to share 64KB of DMA-able memory set aside at initialization. The reason for adding yet another kernel memory allocator is that no existing Mach facility provided the right combination of being able to run during interrupt service,² allocating physically contiguous DMA-able memory, and being space-efficient for small allocations.

3.4.3 I/O Blocking

Linux uses a small block size for I/O operations. For most block devices, the block size is 1KB. For CD-ROM, it is 2KB. In contrast, Mach's block size is 4KB, the page size. The negative effect of a small block size is ameliorated in part by the fact that the Linux block cache code "clusters" the pieces of a multi-block I/O operation. To cluster means to coalesce physically contiguous blocks into a single "segment," and to form a list of segments into a single I/O command. Such a list is suited to devices that have scatter/gather ability.³

¹In the PC architecture, DMA-able memory must be below 16MB, because the ISA bus has only 24 address lines.

²Mach assumes that interrupts do not change the page list.

³A scatter/gather I/O operation consists of an indication of the direction in which the data should move and a list of segments aligned to some boundary. The device

List formation is done without extra copies, since the segment list itself consists only of pointers to segments.

Clustering reduces the number of I/O operations whenever the device provides scatter/gather. (If the device does not support scatter/gather, each segment is a separate operation.) For this reason, we have ported the clustering code from Linux's block cache into the emulation code.

3.5 Synchronization

3.5.1 Between Driver and Processes

Mach and Linux embody fundamentally different design decisions regarding the (a)synchrony of interaction between device drivers and the higher level software that invokes I/O operations.

In Linux, it is assumed that some process is waiting for each I/O operation. The process formats an I/O request, initiates the I/O operation, then sleeps on the I/O request buffer. When the I/O operation completes, the device driver performs a `wakeup` on the I/O request buffer, and the process resumes. In contrast, in Mach the interaction between the driver and the process that initiated I/O is asynchronous. Mach maintains an "I/O-done list" and an "I/O-done thread." When an I/O operation completes, note is made in the I/O-done list. At some later time, the I/O-done thread is scheduled, removes the completed operation from the I/O-done list, and sends a message to the initiator of the operation.

We made the Linux synchronization method compatible with Mach as follows. The I/O request block is passed to the driver locked. When the Linux driver completes the I/O operation, it calls routine `unlock`. This routine unlocks the I/O request block and calls `wakeup`. We replaced `unlock` with code that unlocks the I/O request block then manipulates the I/O-done list to indicate that the operation is finished. It is guaranteed that the I/O-done list already has a record for the operation because the Mach I/O code placed one there before invoking the Linux driver to do the operation.

3.5.2 Between Driver and Hardware

Mach and Linux differ substantially in how they disable interrupts. Mach masks classes of inter-

either "scatters" a write over the segments or "gathers" a read from the segments. Different devices have different limits on how many segments can be accommodated in a single I/O command.

rupts by using `spl` to set the CPU to one of 8 different interrupt priority levels. Linux does not vary CPU priority. Instead, it will disable individual devices by turning them off at the PIC (programmable interrupt controller); Linux also uses the x86 CLI and STI instructions to disable and enable interrupts entirely. Turning off individual devices is a fine grain of control not available in Mach, and changing Mach to use such control would be prohibitively difficult. Consequently, we emulate a Linux driver turning off a single device by using `spl` to turn off the entire class of devices. We have not observed any problems from this over-masking.

Linux is not designed to run on multiprocessors, so Linux device drivers are not concurrency-safe. Mach, on the other hand, contains locking to permit execution on multiprocessors. To use Linux drivers safely within Mach, the emulation code implements a per-device lock that ensures that calls to any single device are serialized.

3.6 Machine Resources

Linux implements an organized approach to the notorious problem of binding interrupt lines (IRQs), DMA channels (DRQs), and I/O port ranges to devices. There is a central allocator that keeps track of each resource. Well behaved device drivers request these resources and release them when they are finished.

Central allocation prevents conflicts, and is a good idea. Since Mach had no such facility, we simply ported this code into Mach for use by the Linux drivers. We did not bother to install similar code in the Mach drivers, so they are ill behaved with respect hardware resource allocation, and they should not be used at the same time as Linux drivers.

Another worthy facility used by Linux drivers that we ported from Linux into Mach is “automatic IRQ detection.” The purpose of this facility is to automatically discover which interrupt line a jumper-configured controller card is using. It works as follows:

1. The device driver calls the “autoIRQ” facility, which installs special interrupt handlers for every unallocated IRQ.
2. The device driver forces the device to interrupt.
3. The device driver calls the autoIRQ facility to receive a report. A timeout is given as an argument.

4. If the device was configured to use one of the available IRQs, then one of the special interrupt handlers was invoked. The autoIRQ allocates this IRQ to the device and indicates so when the device reports asks for the report.
5. If the device was not configured to use one of the available IRQs, then the interrupt was handled by some other device’s interrupt handler and presumably treated as an error. The device driver’s call to autoIRQ for a report times out and indicates that no IRQ was allocated.
6. A side effect of autoIRQ giving its report to the device driver is that the special interrupt handlers are uninstalled.

4 Evaluation

Of the 53 emulated drivers, we tested seven and measured the performance of two, the SMC “Ultra” Ethernet controller and the Adaptec 1542C SCSI controller. Both devices attach to the ISA bus. All the remaining Linux drivers compile. This is not a trivial statement, given that compilation means that emulation code exists for every external name in all drivers.

In the two cases, we compared the performance of the emulated driver versus the native Mach driver.

4.1 Experimental Setting

Our test platform was a Pentium 90MHz system with 16MB of DRAM, ISA and PCI I/O buses, a Fast SCSI-2 disk, and an unloaded Ethernet. Artificial workloads were generated by simple programs we wrote to open a specific device (network or disk) and then access the device via direct calls to the microkernel. In fact, the exact characteristics of the hardware platform and load generation software are not important, for a few reasons. First, we are concerned with latency rather than throughput, so there is no need for the workload generator to be able to run the timed sections often. Second, we are concerned with the relative latencies of two drivers; absolute times are not important. Third, the timed code sections are short and contain few sources of variability; page faults are guaranteed not to happen during a timed section, and no I/O operation occurs in

three of the four tests.⁴ It is not worrisome if the system calls generated by the workload exhibit variable latency; what matters are the short timed sections deep within the kernel.

In order to generate precise timing numbers for short code sections, we used the Pentium's RDMSR instruction. This instruction can read any two of about 40 registers that count the number of certain actions (e.g., cache misses, time ticks) since CPU reset. The time register is a 64-bit counter that is incremented on the edge of every CPU (on-chip) clock signal; i.e., for a 90MHz processor the counter is incremented 90 million times a second. In order to interpret the counter one must know the CPU clock rate; we took code from FreeBSD 2.0.5 that figures this out. The RDMSR instruction itself takes 6 clock ticks. The time to move the returned value to a memory location pushes the overall time to sample the counter up to 8 clock ticks if the memory location is cached; the time could be considerably longer if the location is not cached. On a 90MHz Pentium, 8 clock ticks is about one tenth of a microsecond. Since the shortest code paths we measured is 2 microseconds, we deem the error introduced by counter sampling to be negligible.

Below are two sections, one for each comparison. In each section, we present latency figures in a table and point out and explain any results that are unexpected or interesting.

4.2 Network

DRIVER	SIZE	MIN	VAR
Linux	60 bytes	74 usec	9
Mach	60	62	3
Linux	256	159	22
Mach	256	214	3
Linux	1500	712	31
Mach	1500	582	31

Table 1: Network Transmit Latency (vs. Mach)

We measured the minimum latency and latency variance of the two performance-critical operations: network transmit and receive.

The first time sample was taken at the point where the Linux and Mach drivers first differ. Similarly, the second time sample was taken where

⁴SCSI read and write, and network reception. The timed segment for network transmission includes the I/O.

DRIVER	SIZE	MIN	VAR
Linux	60 bytes	86 usec	12
Mach	60	84	2
Linux	256	265	9
Mach	256	295	18
Linux	1500	655	54
Mach	1500	749	185

Table 2: Network Receive Latency (vs. Mach)

the drivers first re-converge. For transmit, the first point is where the packet is queued and a software interrupt scheduled. The second point is the routine to deallocate a packet once transmission is complete. For receive, the first point is in the packet received interrupt handler, while the second point is in the routine that delivers the packet to the kernel. The transmit path includes the I/O to the Ethernet chip. I/O is finished by the time the receive path begins, but the receive path includes copying the packet from the controller.

For three of the six comparisons (transmit 256, receive 256, and receive 1500), we have the puzzling result that the emulated Linux driver is faster than the native Mach driver. This should not happen since, despite the differences between the drivers, the emulated Linux case always performs strictly more work than the native Mach case. In the two receive cases, we have determined that the entire time difference is due to the single instruction that copies the packet from the I/O controller to DRAM. However, we have been unable to determine what hardware effect is causing the difference⁵ or why this hardware effect occurs in the native driver but not in the emulated Linux driver.

4.3 SCSI

We measured the minimum latency and latency variance of the two performance-critical operations: SCSI read and write. In these tests, two code paths were timed. Path CMD is from the write system call until the command is issued to the host adapter. Path INT is from the command complete interrupt until the I/O request block is retired from the driver's queue of commands. The I/O operation takes place between the first and

⁵We suspect a cache effect, but even with the considerable help of the RDMSR instruction we have not been able to pinpoint the cause.

DRIVER	SIZE	PATH	MIN	VAR
Linux	512	CMD	86 us	1
Mach	512	CMD	29	4
Linux	512	INT	43	4
Mach	512	INT	2	1
Linux	4K	CMD	119	2
Mach	4K	CMD	30	4
Linux	4K	INT	52	1
Mach	4K	INT	2	1
Linux	8K	CMD	141	2
Mach	8K	CMD	35	5
Linux	8K	INT	58	2
Mach	8K	INT	3	0
Linux	16K	CMD	160	2
Mach	16K	CMD	39	4
Linux	16K	INT	74	3
Mach	16K	INT	3	0

Table 3: SCSI Read Latency (vs. Mach)

DRIVER	SIZE	PATH	MIN	VAR
Linux	512	CMD	85 us	2
Mach	512	CMD	29	3
Linux	512	INT	43	4
Mach	512	INT	2	0
Linux	4K	CMD	120	1
Mach	4K	CMD	30	4
Linux	4K	INT	51	2
Mach	4K	INT	2	0
Linux	8K	CMD	143	0
Mach	8K	CMD	35	4
Linux	8K	INT	59	1
Mach	8K	INT	3	0
Linux	16K	CMD	162	0
Mach	16K	CMD	37	5
Linux	16K	INT	74	3
Mach	16K	INT	2	0

Table 4: SCSI Write Latency (vs. Mach)

second timed paths, and hence is not included in any of the timings. As with the network tests, the beginning and ending of the timed paths are the points at which code paths in the Linux and Mach drivers diverge and converge, respectively.

Unlike the network timings, the SCSI results are easily explained. First, for a common driver and data size, the times for read and write are virtually identical. This is to be expected because the two operations are the same except for the direction of data transfer. Second, the numbers for the emulated Linux driver rise with increasing data size. (In contrast, the CMD numbers for the native Mach driver do rise with data size, but very slowly; the INT numbers don't rise at all.) The explanation is that two forms of emulation processing are proportional to transfer size. One is that the emulation code tries to coalesce a multi-block transfer into "segments" with physically contiguous pages. This affects both CMD and INT paths. The other is that the CMD path checks if a DMA operation is scheduled for a memory location beyond 16MB and, if so, allocates "bounce buffers" to compensate.⁶ The Mach driver makes no such check, requiring that DMA be to/from addresses below 16MB.

⁶A bounce buffer is a DMA-able region of memory used as a waystation by a DMA operation whose target address is beyond 16MB.

4.4 Conclusion

Although it is disappointing that we are presently unable to explain the network timings, one point is clear, and that is that the cost of emulation is very low. The case where emulation imposes the highest cost is writing 16KB to SCSI, and here the cost is less than 0.2 milliseconds⁷ for an I/O operation that requires several milliseconds.

In fact, we added considerably more function and overhead to a re-implementation after seeing, in our initial implementation, how little cost was imposed. I/O drivers seem to be an especially appropriate place for emulation since, so long as the I/O bus is slow and the CPU is fast, a considerable number of instructions can be executed by an emulator without having noticeable impact on performance.

5 Summary

Reactions to this work at its outset included expressions that a practical emulation of Linux drivers in Mach would be quasi-miraculous. About quasi-miracles Samuel Johnson once wrote "*A dog's walking on his hind legs [is] not done well; but you are surprised to find it done at all.*" In our case, the dog actually walks quite well.

⁷197 microseconds = 125 on the CMD path plus 72 on the INT path.

We have demonstrated the practicality of incorporating unmodified device driver source code of one operating system into another. Both Mach and Linux are intellectual descendants of UNIX, but they do not share code ancestry. Mach's device drivers are for the most part descended from 4.3 BSD, while Linux device drivers were written from scratch using different assumptions about some important aspects of the surrounding operating system. Nevertheless, the performance penalty for emulation is very limited. In the tests we did, the degree depends on both device and operation, and varies from 2 to 197 microseconds.

We are aware of no related work that shares the essential feature of our work: incorporating unmodified source code of one operating system into another. Of course within the last few years there has been a good deal of work in the UNIX community on emulating the system calls of the far more popular DOS and Windows interfaces; examples include Mach's DOS server [2], Linux "DOSEmu" and "Wine" projects, and a number of commercial efforts such as "DOSMerge" and "WABL." The difference between those efforts and ours is that we include another system's source code into our kernel, so we are emulating intra-kernel variables and interfaces rather than the more standardized and carefully considered system call interface.

The obvious direction for future work in this area is to extend the emulation to include more drivers from Linux and to include drivers from other operating systems. The true mother lode of device drivers in the PC world is of course the DOS or Windows binaries that ship with nearly every piece of peripheral hardware. Emulating these would be enormously more difficult than what we have done so far because of lack of access to source code and because of the huge range of actions taken by these drivers, based on the assumption that they can control the whole machine. Nevertheless, a reasonable starting point might be to emulate network drivers because they are so simple and because so many conform to the NDIS specification [1].

6 Acknowledgements

This work was supported in part by the Advanced Research Projects Agency, ARPA order number B094, under contract N00014-94-1-0719, monitored by the Office of Naval Research; and in part by the Center for Telecommunications Research, an NSF Engineering Research Center supported

by grant number ECD-88-11111.

References

- [1] S. Dhawan.
Networking Device Drivers.
Van Nostrand Reinhold, New York, 1995.
- [2] G. Malan, R. Rashid, D. Golub, and R. Baron.
DOS as a Mach 3.0 Application.
In *Proc. USENIX Mach Symp.*, USENIX, pp. 27–40, November 1991.
- [3] M. Yuhara, B. N. Bershad, C. Maeda, and J. E. B. Moss.
Efficient Packet Demultiplexing for Multiple Endpoints and Large Messages.
In *Proc. 1994 Winter USENIX Conf.*, USENIX, pp. 153–165, January 1994.

7 Author Information

Shantanu Goel is an MS candidate in the Computer Science Department at Columbia University. His research interest is operating systems.

Mailing address: 450 Computer Science Bldg., Columbia University, 500 West 120th St., New York NY 10027. Email address: goel@cs.columbia.edu.

Dan Duchamp is an Associate Professor of Computer Science at Columbia University. His current research interest is the various issues in mobile computing. For his initial efforts in this area, he was named an Office of Naval Research Young Investigator for the period 1993-1996.

Mailing address: 450 Computer Science Bldg., Columbia University, 500 West 120th St., New York NY 10027. Email address: djd@cs.columbia.edu.

Calliope: A Distributed, Scalable Multimedia Server

Andrew Heybey

Mark Sullivan

Paul England

Bell Communications Research
Morristown, NJ 07960

Abstract

Calliope is a distributed multimedia server constructed from personal computers. Preliminary performance measurements indicate that Calliope can be scaled from a single PC producing about 22 MPEG-1 video streams to hundreds of PCs producing thousands of streams. The system can store both variable- and constant-rate video and audio encodings and can deliver them over any network supported by the underlying operating system. Calliope is cost-effective because it requires only commodity hardware and portable because it runs under Unix.

1 Introduction

A multimedia server records and plays data that has real-time delivery requirements. Examples of multimedia data include video and audio streams; some typical applications that might use a multimedia server are video mail, video bulletin boards, and video databases. Two crucial requirements of a multimedia server are that it should be low cost and that it should scale up as demands on the system increase. This paper describes the design and performance of Calliope, a distributed multimedia server constructed from personal computers. Calliope achieves low cost by running on commodity PCs and is easy to scale up by virtue of its distributed architecture.

A traditional network file server cannot be used as a multimedia server for several reasons. First, multimedia data has timeliness constraints. Clients have limited buffering, so data that arrives too late will result in an interruption in audio or a still frame; data that arrives too early will overflow the buffer and be discarded. Traditional file servers make no delivery guarantees.

Second, a traditional file system is not optimized for a multimedia workload. Multimedia clients are willing to accept relatively long delays when a read or write begins, but very little jitter when a large file is being accessed.

Calliope uses several strategies to solve these problems. Timeliness constraints are addressed by dividing the system into real-time and non-real-time components so that data delivery deadlines can be met without using a hard real-time operating system. One or more *Multimedia Storage Units* (MSUs) handle the real-time services while a single *Coordinator* machine handles the non-real-time functions (see Figure 1). The MSUs send and receive multimedia data, manage disk storage, and process simple VCR commands from clients. The Coordinator maintains bookkeeping information, serves as an initial point of contact for users, and schedules requests at MSUs. For very small installations, the Coordinator and MSU software may run on the same machine. Larger Calliope installations still have a single coordinator, but add more MSUs as storage requirements or user bandwidth requirements increase. To achieve as much performance as possible using relatively low performance machines, the MSUs are optimized to handle multimedia files. The MSU has a large-block file system to store large, sequentially-accessed files and its process architecture is oriented toward efficiently moving large quantities of data through the system from disk to network interface (or vice versa for recording).

Calliope can record and play back in real-time both constant bit-rate streams, such as MPEG, and variable bit-rate ones, such as those used by the MBone tools [5]. Since there are many different network protocols and audio/video coding

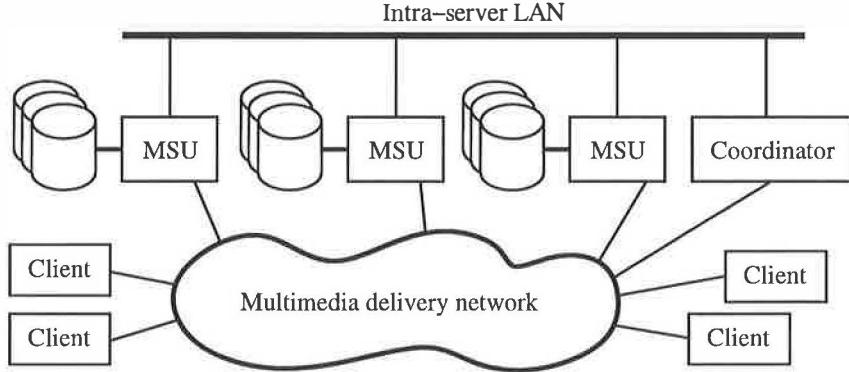


Figure 1: Calliope System Architecture.

standards, Calliope is extensible. Simple modules can be added if necessary to handle different network packet formats and to extract timing information from new encodings.

Calliope does not rely on special-purpose hardware or a hard real-time operating system. The major advantage of a software-based approach running under a standard operating system is portability. As performance of commodity computers improves, Calliope can continue to run on the most cost-effective platform.

The performance results presented in this paper show that twenty-two 1.5 Mbit/sec MPEG-1 streams can be transmitted to an FDDI network from a single Pentium PC running the Calliope MSU software. By combining MSUs, our measurements suggest that Calliope could scale to hundreds of PCs and thousands of customers. We think that such a distributed architecture is appropriate for large scale multimedia servers, but low-end machines have both performance and engineering difficulties that we never expected. In the remainder of the paper, we describe the system architecture of Calliope, measurements of server performance, and some of the PC-related problems we encountered while building Calliope.

2 System Architecture

Figure 1 shows a Calliope installation with three MSUs. The intra-server network connecting the Calliope components is a relatively low-bandwidth network such as Ethernet. The multimedia delivery network that connects Calliope to its clients is a higher-bandwidth network such as FDDI or ATM. If necessary, a Calliope instal-

lation could eliminate the intra-server network and use the multimedia delivery network to carry both intra-server and client-server traffic. This would reduce the number of clients served by each MSU, since some network interface bandwidth would have to be dedicated to MSU control messages. In Calliope, the Coordinator and MSUs communicate using TCP connections. Clients use TCP connections to communicate control information to Calliope and UDP for real-time data delivery. MSUs do not communicate with one another.

The subsections that follow describe Calliope's components and some of its applications. We start with a section on the applications to show what kind of services the system supports. Next, we describe the ways in which the Coordinator allocates resources to provide those services. Finally, we describe the storage management and real-time features of the MSU.

2.1 The Client Interface

We have experimented with several different kinds of client applications for Calliope's multimedia services. Some are fairly simple video-on-demand programs that browse Calliope's table of contents, select content to play, and issue simple VCR commands. We have also used Calliope in a simple video mail application and have used it to record MBone presentations. Finally, we have experimented with an application that keeps simple indices of recorded seminars. Users can examine the index and skip to the portion of the seminar that interests them.

Our primary clients have been Unix™ workstations and PCs running the MBone teleconferencing tools. We have also used Windows™

PCs with hardware-assisted MPEG and JPEG decoders. In one application, we have treated a PC-based MPEG client as dumb set-top box that accepts raw MPEG over UDP. So far, we have not programmed any commercial set-top boxes to act as Calliope clients.

To begin using Calliope, a client establishes a session with the Calliope coordinator. The client can then request a listing of available content, play existing content, or record new content. With appropriate permissions, the client can delete an item of content or make other administrative changes to the server configuration.

The Calliope Coordinator associates a *content type* with each item in its table of contents. Calliope uses the content type to determine the rate at which the content is to be played. Content type also tells whether the content is played at constant or variable rate. Types may be composite; for example, we have a VAT [17] audio type, an RTP (the Internet Real-time Transport Protocol [13]) video type and a *Seminar* type composed of one VAT and one RTP stream. In the current implementation, clients may not define new types without the help of a system administrator to update the Coordinator's internal databases.

Before sending or receiving multimedia content, the client must create a UDP socket and register that socket with Calliope as a *display port*. In Calliope, display ports associate a string name, a content type, and the socket's IP address and port number. The display port need not be on the same machine as the client program that creates it. The software that owns the port socket can be a software encoder/decoder that is part of the client application or a simple driver for a hardware device.

Display ports for composite types can be constructed from previously-registered display ports of the component types. In our example above, a Seminar display port is composed of display ports for RTP and VAT. A client can register many display ports of the same type and unregister them as necessary, but all ports are associated with a single client-Coordinator session. When this session is dropped, the Coordinator deallocates its local representation of the ports.

To read an item of Calliope's content, the client sends a request containing the content name and the name of a display port. Calliope checks that the port and the content have the same type and assigns the resources required to play the content. When it is ready to play the content stream, Cal-

loope creates a control connection to the client on which the client can send VCR commands: pause, play, seek, and quit. For some content items, clients will also be able to issue fast forward and fast backward commands (this is described in more detail in Section 2.3.1). Recording on a Calliope server is similar to reading, but the client request must also contain an estimate of the recording length.

2.2 Coordinator Architecture

The Coordinator is the global resource manager for Calliope. It maintains a small administrative database and a set of scheduling queues. The database contains information about customers, content stored on Calliope, and resources owned by the system. The Coordinator uses the database to tell what MSUs are available, how many disks each one has, and how much disk space remains unused. The Coordinator also keeps track of which items of content are stored on each disk and what content types are available. As the previous section stated, each item of content has a type. The content type entry contains a *bandwidth consumption rate* which gives the expected rate at which content of this type is to be played and recorded.

The Coordinator uses its databases to authenticate clients, satisfy their requests for the table of contents, and to allocate resources as clients play or record content. When Calliope receives a read request, the Coordinator finds an MSU with a disk that both contains the requested content and has enough bandwidth available to satisfy the request. As the Coordinator assigns resources to clients, it keeps track of load by processor and disk. If a client's request cannot be satisfied, the Coordinator queues the request until an MSU with the necessary resources becomes available.

The Coordinator informs an MSU of a scheduling decision by sending a message describing the client's request. Once the request is scheduled, the client does not communicate with the Coordinator until the stream is terminated. VCR commands between the client and Calliope go directly to the MSU. As soon as it is ready to deliver the content stream, the MSU establishes a control stream (TCP connection) with the client for these commands. After a "quit" command from the client, the MSU informs the coordinator that the stream has been terminated.

If a client requests a composite content type,

the Coordinator creates a *stream group* to play the content. A stream group has one member for each of the atomic subtypes of the composite type. For example, if a client played an item of the composite Seminar type described in the previous section, the RTP video and VAT audio subitems would become a two-item stream group. All streams in a stream group are controlled by the same VCR commands. Calliope assigns all streams in a group to the same MSU so that the client's commands can start and stop all streams simultaneously. Synchronizing the streams would be difficult if streams from the same group were assigned to different machines.

When a client records content on a Calliope server, the Coordinator must allocate disk storage as well as bandwidth. The client write request includes an estimate of the length of the recording (in seconds). The Coordinator uses this estimate and the content type information to determine how much disk space the recording will consume. It must schedule the request on an MSU that has both disk space and bandwidth available. If the client overestimates the length of the recording, the unused space will be returned to the system once the recording session has completed.

To support variable rate content, the content type table contains separate rates for disk space and bandwidth consumption. A constant-rate stream will consume disk bandwidth and disk space at the same rate. For data with a variable-rate encoding, the system should allocate bandwidth more conservatively than disk space. The bandwidth consumption rate should be closer to the stream's peak rate and the storage consumption rate should be closer to the average rate.

Calliope's Coordinator has some support for fault tolerance. The Coordinator detects when one of the MSUs fails by a break in the TCP connection between the coordinator and the MSU. When an MSU is down, the Coordinator marks it as unavailable in the scheduling database. When the MSU becomes available again, it contacts the Coordinator and is restored to the scheduling database. Calliope does not recover from Coordinator failures.

2.2.1 Disk and Network Scheduling

Calliope's MSUs use double buffering and careful disk and network device scheduling to ensure that data is delivered at a regular rate and peripheral devices are fully utilized. In the case of a read, double buffering means the network process

is sending data to a client from one main memory buffer while the disk process is loading the other. When the network process empties its buffer, the disk process should already have filled the other buffer. The two processes swap buffers for the next cycle.

To allocate bandwidth of a single disk, we give the disk a *duty cycle* which is divided into *slots*. Each slot is long enough to read or write a single disk block for one client stream. The number of slots in a cycle is the maximum number of block transfers that can be accomplished during the time it takes for a single stream to transmit its block. In other words, the cycle must be short enough that each client's disk transfer completes before the network transfer runs out of data to transmit.

In order to replay variable-rate data packets at the correct times, the network process constructs a *delivery schedule* as the data is recorded. The schedule associates delivery times with data packets. When variable-rate data is replayed, the network process uses the stored delivery schedules to determine when packets are delivered to the network. For constant bit-rate streams, the delivery schedule is calculated rather than stored. The arrival times in delivery schedules are not absolute; they are offsets from the beginning of the recording session.

When it stores the delivery schedule and data on disk, Calliope interleaves them in a single file using a data structure similar to a primary B-tree [4]. In a primary B-tree, the file blocks contain both data and a search tree. The top parts of the search tree are stored in internal pages containing only keys and pointers to lower-level pages. The lower parts of the tree are stored in leaf pages with the data. In our case, the key for the search tree is delivery time. A sequential scan of the B-tree gives the data packets in the order they must be delivered to the network.

Calliope's variant on B-tree is called Integrated B-tree (IB-tree) because it integrates the internal pages into the data pages. Calliope's large disk blocks allow it to make internal pages smaller than leaf pages without increasing the height of the tree in most cases. We use 28 KByte internal pages (with 1024 keys) and 256 KByte data pages. As the B-tree is constructed, Calliope creates the internal pages and data pages in a manner similar to normal B-tree construction. When an internal page fills up, it is copied into the current data page instead of being written separately

on disk. During seeks, Calliope traverses the internal pages of the search tree in the usual way. During sequential reads, the internal pages are read in as part of the data page but ignored. They are so small and only appear in 0.1% of the data pages so they do not affect read bandwidth appreciably. On writes, the IB-tree writes both data page and internal page using a single disk transfer and seek. If the two pages were stored separately, the internal page writes would add slots to Calliope's disk duty cycle and the extra seeks would reduce disk utilization.

Finally, Calliope does not use a real-time operating system and FreeBSD timers have only 10 ms granularity, so delivery times are only approximate. While timer granularity will introduce jitter, clients will have to be able to handle the jitter introduced by the multimedia delivery network anyway. We assume that clients have enough buffer space to smooth any jitter introduced by either the approximate scheduling or the intervening network. A 200 KByte buffer will hold more than one second of 1.5 Mbit/sec video. Calliope will not add more than 150 milliseconds of jitter in the worst case (see Section 3.2.1) and any network that introduces more than 850 milliseconds of jitter is probably not usable for video delivery.

2.3 MSU Software Architecture

Each MSU is a PC with a set of disks, an interface to Calliope's intra-server network and an interface to the external high-speed network. The MSU runs a simple multi-process control program that assigns a process to each network device and disk while a central process handles RPCs from the Coordinator and from clients. The MSU processes must communicate in order to share resources and to start and stop streams. Instead of using expensive semaphore operations, the MSU processes communicate using a shared memory queue structure that relies on the atomicity of memory read and write instructions to produce atomic enqueue and dequeue operations.

When the client starts a read stream, the MSU's disk process loads data from disk into a shared memory buffer. The network process then packetizes the buffer and sends it out through the high speed interface. The network process ensures that packet delivery proceeds on schedule. The disk process makes sure that the network process always has buffered data ready to send. When data is recorded, the network pro-

cess fills buffers and the disk process writes full ones to disk. The description of Calliope's performance in Section 3 contains more detail about the MSU data path.

2.3.1 Fast Forward and Backward

The MSU cannot send a data stream to clients at a higher rate than the one at which it was received. To implement fast forward and fast backward scans, we used an offline filtering program. Thus far, we have only implemented filtering for MPEG streams. The filtering program reads the recorded stream, selects every fifteenth video frame, recompresses the filtered stream, and loads it into the server. For the fast-backward version, the frames are stored in the filtered stream in reverse order. This filtering procedure is not automatic in the current implementation; an administrator has to produce the fast forward and fast backward versions of the content.

An administrative interface is used to load the fast forward and fast backward files into the server in a way that allows the server to associate the files with the fast forward and fast backward VCR commands. The MSU remembers which files contain the normal rate, fast forward, and fast backward versions of the same content. If a client issues a command to switch from normal rate to fast forward, the MSU seeks to the frame in the fast forward file corresponding to the current frame of the normal rate file. The user will experience a few seconds of delay as the MSU waits for the client's next disk slot, reads the required block of the fast forward file into memory, and starts delivering frames to the network device. Switching back to normal rate follows the same procedure.

We briefly considered implementing a dynamic fast forward and fast backward mechanism that extracted the fast rate streams directly from the normal rate stream. In such a scheme, the MSU would use the schedule information to skip over frames, delivering only selected ones to the network. This scheme was impractical for two reasons.

First, if the data stream uses inter-frame compression, some frames may not be safely skipped. In inter-frame compression, decoding a given frame requires information in other frames around it in the stream. If the stream uses intra-frame compression or no compression, the MSU could send selected frames to the user. MPEG uses a combination of inter-frame and intra-frame

compression. Most frames are inter-encoded, but intra-encoding is used for every N-th frame, where N is a parameter determined at the time of encoding (typically, fifteen to thirty). The MSU could implement fast forward for MPEG streams by sending selected intra-encoded frames. However, the MPEG encoders that we have produce an opaque stream with no framing information. While recording, the MSU would have to search the stream to find the intra-coded frames. Parsing the MPEG stream is too expensive to do in real time.

The second reason we did not implement this scheme is that it would make disk scheduling hard. Skipping frames allows the MSU to send a fast forward stream using the same network resources as a normal-rate stream. However, fast forward delivery has a larger impact on disk usage than normal rate delivery. If the MSU reads from disk only the frames that it will send in the fast forward stream, it has to issue many small read requests instead of a few 256 KByte ones. This will significantly worsen disk performance. A more practical approach is to read all of the stream's frames from the disk and then skip over the unneeded frames once they are in memory. However, in this case, the MSU must read fast forward streams from disk at several times the normal stream rate. Allowing users to switch back and forth between high-rate and low-rate disk reads would complicate our scheduler.

2.3.2 MSU Extensibility

The MSU is designed so that support for new protocols can be added to the system easily. A “protocol” in this context is something no more than complex than (for example) RTP—essentially a header definition and a few control messages. The MSU currently supports RTP [13], VAT [17] audio, some home-grown protocols and any protocol and/or encoding which can be handled by transmitting fixed sized packets at a constant rate.

An MSU *protocol extension module* is comprised of two functions. The first performs any operations required by the protocol beyond the normal sending or receiving of data packets. For example, the RTP protocol uses two ports—one for control messages and one for data. The RTP module for the MSU manages the control socket. During recording, the RTP module interleaves the control messages with the rest of the data stream before the data is given to the disk process. On output, the opposite process is per-

formed. The MSU calls the second extension function during recording to construct a delivery schedule. This function creates a delivery time to use when the incoming packet is replayed. By default, the MSU derives the delivery time from the packet's arrival time. If there is a timestamp in the protocol's header, then a protocol extension function may derive delivery time from the timestamp. Using the sender-generated protocol timestamp instead of the packet's arrival time has the advantage that it does not include the effects of network-induced jitter.

2.3.3 MSU File System

The MSU has to manage files that are often large (a two hour MPEG-1 movie is 1.35 GByte) and are usually read and written sequentially. Instead of the BSD fast file system, the MSU uses a simple user-level file system tuned to the multimedia workload. The MSU file system does its own memory management and uses raw disk I/O to avoid user-to-kernel-space copying.

Instead of a block cache, available main memory is organized into large buffers to support read ahead and write behind. Large disk blocks and large buffers mean that large amounts of data can be transferred per disk access, lessening the performance impact of disk seeks. With 256 KByte transfers, the MSU achieves 70% of the maximum disk transfer bandwidth. Large file block size also decreases the size of the file system meta-data to the point that it can be entirely cached in main memory. An LRU block cache would impair performance because there is not enough data locality or sharing to make the cache effective. Different clients may view the same data at approximately the same time, but a 256 KByte buffer contains only about one second of 1.5 Mbit/sec MPEG-1 video. To share data in a block cache, clients would have to be synchronized to within a second of one another. Without some application-level coordination, this is unlikely to occur. Locality of reference by a single client is also unlikely. Clients tend to access multimedia files sequentially so the client would rarely rereference its own cached data.

The current implementation of the MSU does not employ disk head scheduling. The MSU services the customers for each disk in a round-robin fashion, resulting in random seeks between disk transfers. Further gains in disk performance could be realized by ordering the I/Os to minimize seek distances. We may eventually imple-

ment disk head scheduling, but we do not expect large performance improvements for two reasons. First, disk rotation and head settling times contribute a substantial fraction of the total seek time; disk head scheduling will not have any effect on these. Second, the MSU's large block size already limits the effects of seeks on disk performance. Using a simple program that simulated 24 concurrent users reading random 256 KByte disk blocks, we found that an elevator scheduling algorithm improves throughput by only about 6% for our disks.

In the current implementation, Calliope's MSU does not stripe files over its disks. When a client writes a file, all blocks go to a single disk. It would be easy to lay out a file so that consecutive blocks are on "adjacent" disks. The disk process in this case would read or write blocks from its disks in a round-robin fashion.

Disk bandwidth management is a little more complex in this case. The duty cycle must cover all N disks and contain $D * N$ slots, where D is the number of slots in a single disk's duty cycle. When a client arrives at the MSU, it is allocated a disk slot and must wait at most $D * N - 1$ slots before the MSU begins to deliver data.

The advantage to this kind of disk management is that we can still utilize the disks well even if workload is unpredictable. If an MSU has N items of content striped across N identical disks, all of the system's customers can access any of the N items. If each of the items were on separate disks, only $1/N$ of the system's customers can access any one item of content. In the non-striped case, we can make copies of popular content on several disks, but we must anticipate usage trends in order to choose the content to copy. We must also use additional disk space to get additional disk bandwidth.

One disadvantage of striping is that the client must delay every time it issues a VCR command while a disk slot becomes available. As in the non-striping case, the client waits at most as long as the duty cycle. If the MSU file system used striping, this delay is N times as long as it is in the non-striped case. We initially felt that this delay would be unacceptable to our users. In retrospect, we were probably wrong.

The second disadvantage is that management of streams with different fixed rates or a combination of variable and fixed rates is harder in a striped environment. When all streams play at the same rate, controlling the rate at which users

enter the system ensures that none of the disks becomes overloaded. If different files are consumed at different rates, the block size must vary by file or clients will progress across the disk at different rates depending on the content they are playing or recording. Variable block sizes make file system implementation more challenging. Allowing users to move from disk to disk at different rates would allow disks to become oversubscribed.

3 Performance

To demonstrate the scalability and capacity of the system, we have run several kinds of performance measurements. First, we measured the throughput capacity of our hardware by running simple programs that move data through the same data path as the MSU. These measurements give an approximate upper bound on the performance of the MSU for our hardware. Second, we estimated the MSU's throughput by observing how well packet delivery deadlines are met when the system delivers varying numbers of constant-rate and variable-rate streams. Finally, we measured the load on the internal network and Coordinator due to client requests in order to determine how many MSUs can be combined into a single system.

In our experiments, the MSU software runs under the FreeBSD [7] operating system version 2.0.5 on a 66 MHz Pentium PC from Micron Computer Corporation. The Micron has two busses: a fast PCI bus and a slower EISA bus. Each MSU contains one or more Buslogic EISA bus fast-differential SCSI host adaptors each having one or more 2 GByte Seagate Barracuda disks, 32 MBytes of main memory, a single SMC ISA bus Ethernet interface to the internal network and a DEC DEFPA PCI bus FDDI interface to the high speed network.

3.1 Baseline Measurements

In order to estimate the maximum potential throughput of Calliope, we measured the performance of several simple programs exercising memory, disks, and network interface. Since these programs perform almost no computation, they are measuring the speed at which the PC hardware and operating system can move data from disk through memory and out the FDDI interface.

The baseline tests use one process per device as the MSU does. The disk process is a sim-

	FDDI only	Disks only			Disks and FDDI			
		Disk 1	Disk 2	Disk 3	FDDI	Disk 1	Disk 2	Disk 3
0 disk	8.5							
1 disk (one HBA)		3.6			5.9	3.4		
2 disk (one HBA)		2.8	2.8		4.7	2.4	2.4	
2 disk (two HBA)		2.9		2.9	2.3	2.7		2.7
3 disk (two HBA)		2.2	2.2	2.7	1.4	1.9	1.9	2.5

Table 1: Baseline Performance Measurements. The measurements show the throughputs of simple test programs writing to the FDDI interface and reading from the disks (in MBytes/sec). The table is divided across the top into three groups of experiments: FDDI only, disks only, and both FDDI and disks running simultaneously. Within each group, each row of the table shows the performance of each component that took part in a particular run. Tests labeled “one HBA” (one SCSI Host Bus Adaptor) had all disks on the same SCSI chain; those labeled “two HBA” had at least one disk on each of two SCSI chains. See the text for a detailed description of the test programs used.

ple program that performs 256 KByte reads of the raw disk device at random offsets. The network process is a slightly modified version of the ttcp [16] program, which sends data from memory to a peer process on a machine across the FDDI network. We changed ttcp so it did not touch the data before sending (since the MSU network I/O process (IOP) does not touch its data). We also changed it to send successive segments from a large buffer instead of repeatedly sending a smaller buffer. If a small buffer is used, ttcp reports artificially high performance because the buffer stays in the processor cache, speeding up the user-to-kernel-space copy. The arguments to ttcp were:

```
ttcp -t -u -s -l 4096 -L 1m -n 100000 <dest>
-t          Transmit mode.
-u          Use UDP instead of TCP.
-l 4096    Send 4k packets (default is 8k).
-n 100000  Send 100000 packets.
-s          Send from memory, not stdin.
-L 1m      Step through 1MB buffer while
           sending (we added this option).
```

Note that, while ttcp misreports network bandwidth for transmitting UDP packets under some versions of Unix, it is accurate when run on FreeBSD. Some systems silently discard packets when the interface output queue is full, leading ttcp to report inflated performance numbers. FreeBSD returns ENOBUFS when the interface output queue is full. Ttcp then sleeps briefly and tries to send the packet again.

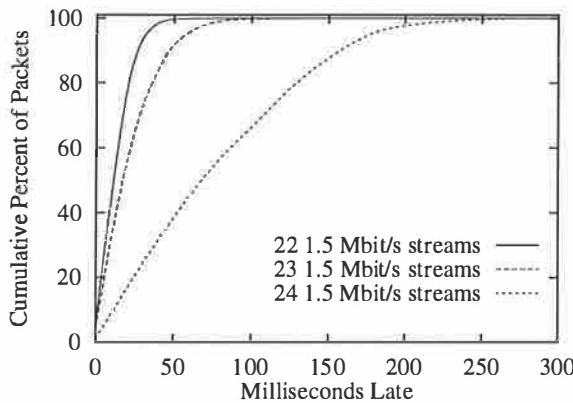
Table 1 displays the results of the baseline measurements. According to the measurements, the highest throughput attainable with our particu-

lar hardware and operating system combination is 4.7 MByte/sec¹. This corresponds to the “2 disk (one HBA)” experiment in which the FDDI wrote 4.7 MByte/sec, and the 2 disks read 2.4 MByte/sec each. With only one disk running, the FDDI can go faster, but the disk does not then produce enough data to keep up with the network.

It is surprising that we could not achieve better performance with multiple SCSI host bus adaptors (HBAs) running simultaneously. As Table 1 shows, the FDDI performance is dramatically lower when running two disks on two separate HBAs versus two disks on the same HBA, while performance achieved by the disks in this case is only slightly improved. We believe that this effect is a hardware problem either on the part of our motherboard or the HBAs.

We first became aware of the problem when we noticed that the system clock slowed down if multiple HBAs were active. Upon further investigation, we discovered that the system was missing clock interrupts. It turned out that “in” and “out” instructions (which are needed both to service interrupts and to read the hardware timer used in FreeBSD to keep time) could take a very long time when two HBAs were running. Specifically, the sequence of instructions needed to read the hardware timer took approximately 4 microseconds with no disk activity; it occasionally took a millisecond with one HBA running, and often took 20 milliseconds with two HBAs running. To measure this phenomenon, we disabled interrupts and timed the code using the

¹All of the measurements in this section are in 10^6 bytes/sec units.



Graph 1: Cumulative Packet Delivery Distribution of Constant Bit Rate Streams. The curves show the percent of packets delivered within a given number of milliseconds of their deadline. The three workloads depicted show the service provided as the MSU reaches its performance limit.

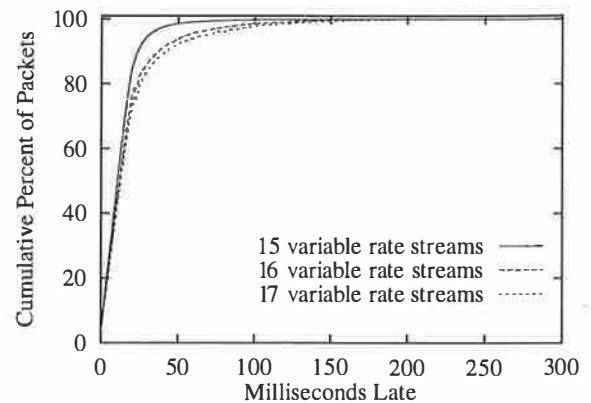
Pentium microprocessor's internal cycle counter. We worked around the bug by changing FreeBSD to keep time using the Pentium cycle counter (so that missed clock interrupts would not affect the time of day, though they might result in timer interrupts being late).

3.2 MSU Throughput

3.2.1 Constant-Rate Streams

Graph 1 shows how closely an MSU with 22, 23 and 24 constant-rate streams keeps to the real-time packet delivery schedule. In each experiment, the MSU delivered streams from two disks on one host bus adaptor for six minutes, and generated approximately 16480 four KByte FDDI packets per stream. The graph's X-axis is the number of milliseconds a packet was sent after its deadline. The Y-axis gives the cumulative percentage of all packets delivered in the experiment that fall in each one-millisecond bin.

The measurements show that an MSU can deliver up to 22 1.5 MBit/sec streams with very good performance for each stream. In this case, only 0.4 percent of the packets are delivered more than 50 milliseconds late and no packets are more than 150 milliseconds late. As more streams are added the quality first degrades gradually and then dramatically. With 24 streams, only 38 per-



Graph 2: Variable Bit Rate Cumulative Packet Delivery Distribution. The curves show the percent of packets delivered within a given number of milliseconds of their deadline for variable rate streams. The streams have average rates ranging from 635 to 877 KBit/sec. The three workloads depicted show the service provided as the MSU reaches its performance limit.

cent of the packets are delivered within 50 milliseconds of their deadline.

This performance is approximately 90% of the baseline performance, indicating that the overhead introduced by the MSU for constant-rate streams is not excessive. The biggest performance problem of the system is the one discussed above in Section 3.1. The current baseline performance might be considerably improved with different hardware.

3.2.2 Variable Rate Streams

Graph 2 shows how closely an MSU with 15, 16, and 17 variable-rate streams keeps to the real-time packet delivery schedule. For these tests, three different files encoded by NV [6] were used (so the files were played multiple times to produce the required number of streams). The three different files used in the test had average rates of 650, 635, and 877 KBit/sec. Note that the performance for variable-rate streams (by this metric) is substantially worse than that of the constant-rate streams. There are several reasons for this performance difference.

First, the packet size of the variable-rate streams is much smaller than the four KByte

packets used for the constant-rate tests. Most of the packets in the streams are about one KByte long. There is four times as much processing overhead for both the MSU code and the kernel networking code.

Second, the video is quite bursty. NV encodes a frame and then sends it out as quickly as possible, resulting in bursts of back-to-back packets. Measured using a 50 millisecond sliding window, the peak rates of the files ranged from 2.0 to 5.4 MBit/sec. It is impossible for the MSU to preserve the exact timings for many streams with bursty traffic on this time scale due the non-real-time nature of the MSU. Packets at the tails of the bursts will be delayed relative to their deadline.

In addition, the clients in the variable-rate tests viewed only three different files. All of the streams in the tests were started simultaneously. This means that every time the data file contained a burst of packets, one third of the streams in the test transmitted the burst at the same time. Thus, the MSU often had more packets to deliver simultaneously than the apparent data rates would suggest and some packets were necessarily delivered late. Furthermore, when tested while transmitting only a single file, the MSU could only produce 11 streams instead of 15. This unrealistic scenario is a limitation of our automated test setup; in practice clients do not start streams in synchrony nor would there be such a limited selection of content.

3.2.3 Bottlenecks

At present, the bottleneck in our system is that we cannot make use of more than one SCSI host bus adaptor simultaneously, limiting the data rate to 4.7 MBytes/sec. The next bottleneck is memory bandwidth. Our system can read memory at 53 MByte/sec, write it at 25 MByte/sec, and copy at 18 MByte/sec. As the MSU reads a file from disk and sends it to a client, the data traces the following path through the memory of the MSU PC:

1. Write (DMA from disk to user memory in the raw disk read).
2. Copy (user space buffer to kernel mbuf in network send).
3. Read (UDP checksum).
4. Read (DMA to FDDI interface).

Therefore, the fastest rate at which our test system could move data along this path (assuming that DMA accesses memory at the same speed as the processor) is:

$$\frac{1}{\frac{1}{25} + \frac{1}{18} + \frac{2}{53}} = 7.5 \text{MByte/sec.}$$

To measure this disk-less data path, we replaced the disk I/O process in our baseline measurements with a process that simply wrote constant values into memory buffers. Using this process setup, the system moved data at about 6.3 MByte/sec (that is, it sent UDP packets at that rate while another process simultaneously wrote memory buffers at the same rate). The difference between the maximum 7.5 MByte/sec and the measured value is due to other memory accesses occurring in addition to the data movement. In particular, we believe that instruction fetches account for much of the difference. All the code necessary for the MSU (and kernel) to move the data will not fit in the first-level instruction cache, and the second-level cache will be flushed by all the data movement.

3.3 Scalability

The Coordinator and internal network are the only shared resources in the system, so their capacity will eventually limit system size. During normal operation, most of the load on the Coordinator and network consists of client requests for new streams and stream termination notifications. In a real system, the maximum request rate depends on the number of MSUs in the system and the length of the average viewing session.

To measure the effect of scheduling requests on shared resource loads, we have created a fake MSU which, when scheduled, delays for 50 ms and then reports that the user has terminated the stream. We start two of these MSUs on different machines and started two clients who together sent 10,000 requests to the coordinator at a rate of about 60 requests per second. We measured the Coordinator's CPU utilization at 14% and the network utilization at 6%, both relatively insignificant loads. Even if sessions are as short as one minute, a large scale implementation of Calliope serving 3000 simultaneous streams (150 MSUs at 20 streams each) would need to service only 50 requests per second.

4 Related Work

Commercial network based video and multimedia servers have become quite common in recent years. Several large-scale distributed video servers — including ones by DEC, Hewlett-Packard, Oracle, IBM, and Microsoft — are under production or currently available. Starlight sells a smaller-scale ethernet-based centralized video server. While some of the commercial systems have designs similar to ours, the focus of the paper is on performance issues and implementation experience which the commercial vendors have not published. Furthermore, many of the commercial systems rely on custom hardware and operating systems to achieve acceptable performance while Calliope uses commodity hardware and OS software for greater portability and lower cost.

We are familiar with three research multimedia servers comparable to ours. Freedman and DeWitt [8] describe simulations of a video-on-demand system in which clients request data as their buffers empty rather than having the server deliver the data at a pre-determined rate. Smith [14] focuses on coding and protocol issues rather than on the architecture and scalability of the server itself. Gelman et al. [9] describes a multimedia server built from custom hardware. Their system includes an end-to-end network distribution system for video-on-demand that uses buffering in switches and specialized telephone line-cards.

Other groups have considered specific aspects of server design, most notably, disk layout strategies. Louher and Shepherd [11] describe a file system for striping multimedia data across several disks with file system meta-data on a separate disk. They concentrate on constant rate streams but can support streams of different rates by varying disk I/O sizes so that reads happen at constant intervals. Kalns and Hsu [10] describe a video server implemented using the Vesta parallel file system on an IBM SP-1 parallel computer. It uses JPEG encoding at a constant frame rate but variable frame size, so some mapping from timestamp or frame to byte offset would be required in order to seek to a given frame or time offset, though the paper does not mention how seeking is implemented. Chang and Zakhor [3] describe a disk layout scheme for storing video encoded using a scalable coding on RAID disks in such a way as to be able to serve as many clients

as possible from the array. The encoding used is constant rate. Vin and Rangan [18] analyze a method of interleaving data on disk to make the most of disk bandwidth but do not address variable-rate streams.

Another class of systems that include support for video include several research and commercial DBMSs [15] [2] [1]. Some of these systems support sophisticated query-by-image-content features, but do not support real-time data delivery — the essential feature of Calliope.

5 Conclusions

In this paper, we have sketched the architecture and performance characteristics of Calliope—a distributed, scalable real-time storage system for multimedia data. Calliope consists of a collection of off-the-shelf PCs running Unix. One machine, designated the Coordinator, maintains a database of the stored content and serves as the point of contact for new clients. The rest of the machines are Multimedia Storage Units (MSUs) which record and play back real-time content for clients. Calliope is extensible in order to support many types of networks, protocols, and audio or video encodings.

We have measured the performance of Calliope running on Pentium-based PCs under the FreeBSD operating system. Our test system can support 22 1.5 MBit/sec simultaneous constant-rate streams or 15 variable-rate streams (at an average rate of 635 to 877 KBit/sec and peaks up to 5.4 MBit/sec). We have shown that inexpensive hardware and a general-purpose operating system can be used to transmit many audio and video streams with good performance.

Acknowledgements

We would like to thank Bob Gray, Peter Bates, Sid Devadhar, Ravi Jain, Ben Melamed, Andy Ogielski, Marc Pucci, Norman Ramsey, and Yatin Saraiya for comments and suggestions.

References

- [1] A. Biliris. The performance of three database storage structures for managing large objects. In *Proc. ACM SIGMOD Conference*, San Diego, California, June 1992.

- [2] M. J. Carey et al. Shoring up persistent applications. In *Proc. ACM SIGMOD Conference*, Minneapolis, MN, May 1994.
- [3] E. Chang and A. Zakhor. Scalable video data placement on parallel disk arrays. In *Storage and Retrieval for Image and Video Databases II*, volume SPIE 2185, 1994.
- [4] D. Comer. The ubiquitous B-Tree. *ACM Computing Surveys*, 11(4), 1979.
- [5] H. Eriksson. MBone: The multicast backbone. *Communications of the ACM*, 37:54–60, 1994.
- [6] Ron Frederick. Experiences with real-time software video compression. In *Proceedings of the Sixth International Workshop on Packet Video*, pages F1.1–1.4, Portland, Oregon, September 1994.
- [7] FreeBSD operating system. Information and source code available via HTTP from www.freebsd.org.
- [8] C. Freedman and D. DeWitt. The SPIFFI scalable video-on-demand system. In *Proc. ACM SIGMOD Conference*, pages 352–363, 1995.
- [9] A. Gelman, H. Hobrinski, L. Smoot, S. Weinstein, M. Fortier, and D. Lemay. A store and forward architecture for video on demand service. In *Proceedings of the IEEE ICC'91*, Denver, Colorado, 1991.
- [10] E. Kalns and Y. Hsu. Video on demand using the Vesta parallel file system. In *Third Annual Workshop on Input/Output in Parallel and Distributed Systems*, April 1995.
- [11] P. Lougher and D. Shepherd. The design of a storage server for continuous media. *The Computer Journal*, 36(1):32–42, 1993.
- [12] Coding of moving pictures and associated audio (MPEG). Int'l Organization for Standardization/Int'l Electrotechnical Institute, September 1990. ISO/IEC JTC1/SC2/WG11.
- [13] H. Schulzrinne, S. Casner, R. Frederick, and V. Jacobson. RTP: A transport protocol for real-time applications. Internet draft: draft-ietf-avt-rtp-07, Internet Engineering Task Force, March 1995.
- [14] B. Smith. Implementation techniques for continuous media systems and applications. Technical Report UCB/CSD 94/845, Computer Science Division—EECS, U.C. Berkeley, December 1994.
- [15] M. Stonebraker and M. Olson. Large object support in POSTGRES. In *Ninth International Conference on Data Engineering*, 1993.
- [16] TTCP network performance measurement program. Available for anonymous ftp from [ftp.sgi.com:sgi/src/ttcp](ftp://ftp.sgi.com:sgi/src/ttcp).
- [17] VAT audioconferencing program. Available for anonymous ftp from [ftp.ee.lbl.gov:conferencing/vat](ftp://ftp.ee.lbl.gov:conferencing/vat).
- [18] H. Vin and P. V. Rangan. Designing a multiuser HDTV storage server. *IEEE Journal on Selected Areas in Communications*, 11(1):153–164, January 1993.

Biographies

Andrew Heybey (ath@bellcore.com) received his BS and MS in EECS from the Massachusetts Institute of Technology. He worked for several years at MIT before joining Bellcore in 1993. His interests include networks, real-time protocols, operating systems, and computer architecture.

Mark Sullivan (sullivan@bellcore.com) received his BS in Math Sciences from Stanford University and MS/PhD in Computer Science from the University of California at Berkeley. Since 1992, he has worked as a parallel computing researcher at Bellcore. His interests include database management systems, networks, fault tolerance, and poetry.

Paul England (england@bellcore.com) holds a BS in physics from the University of Birmingham, and a PhD in physics from Imperial College London. He has been a Research Scientist at Bellcore since 1986 and has worked extensively on the design and characterization of novel semiconducting, superconducting and optoelectronic devices. His current interests include applications and architectures for networked multimedia.

Simple Continuous Media Storage Server on Real-Time Mach

Hiroshi Tezuka[†]

Japan Advanced Institute of Science and Technology

{tezuka,tatsuo}@jaist.ac.jp

<http://mmmc.jaist.ac.jp:8000/>

Tatsuo Nakajima

Abstract

This paper presents the design and implementation of a simple continuous media storage server: CRAS on Real-Time Mach. CRAS is a specially optimized storage system for retrieving multiple continuous media streams such as audio and video from a disk at constant rates for small scale distributed multimedia systems.

Many previous continuous media storage servers have focussed on high throughput for supporting as many video sessions as possible. However, these servers are too big and complicated for playback applications that retrieve continuous media data from the local disks of personal computers. Also, there are many continuous media systems requiring small continuous media storage servers that can be shared by a small number of applications. To reduce hardware costs, the servers should run on less powerful computers. This means that the previous big and complicated servers are not appropriate for such small scale environments.

We show that our simple continuous media server for small scale systems can guarantee the retrieval of continuous media data at a constant rate, and provide high throughput even though it is compact and simple.

1 Introduction

The increasing performance of microprocessors allows us to handle continuous media such as digital audio and video on personal workstations[11]. A continuous media storage system is one of the most important components for distributed continuous media systems. Since audio and video are timing-dependent continuous media, such storage systems should be able to ensure that multiple continuous media streams can be retrieved from disks at constant rates.

[†]He now belongs to Real World Computing Partnership
Tsukuba Mitsui Building 16F, 1-6-1 Takezono, Tsukuba-shi,
Ibaraki, 305, JAPAN.

Several research groups have been building continuous media storage servers for video servers which can retrieve many streams concurrently[1, 5, 12, 13, 19]. The most important goal of these systems is to retrieve as many streams as possible. In addition, these systems focus on the special organization of data blocks on disks to reduce disk seeks, rotational latency, and the overhead of disk head scheduling. Disk striping is also used to increase the throughput of the disk. Since the storage systems require these mechanisms, the servers have included large complicated implementation of directory management, block caching, meta data management, and block allocation.

On the other hand, there are many personal and interactive continuous media applications that require access to local disks or a small shared storage server. For example, a typical application for future workstations is a travel coordinator for two people in different places. They use a map in a shared window on their respective systems, and check interesting sightseeing points in a video database while using a conferencing tool to talk to each other. This application should retrieve video clips at constant rates from the database stored on disks in the user's machines, and these retrievals must avoid interference from concurrent activities contending for the same disk.

Many commercial systems such as QuickTime[4] and Video for Windows[18], used by many commercial continuous media applications, can perform a constant rate stream retrieval easily because only one application at a time is supported, so there is no contending disk activity. However, future continuous media applications will outgrow such single task systems.

Small scale distributed continuous media applications also require simple continuous media storage servers. These servers should be compact and simple to minimize the hardware costs of the computers on which the servers run. Thus, traditional complicated storage servers that require a large amount of memory and powerful CPU may not be suitable for small scale

applications.

Continuous Media Player[14], which runs on Unix, supports the retrieval of multiple movie files from disks. It provides a storage system that is designed for simultaneous accesses by many continuous media applications. The system can be integrated with existing Unix applications easily because it is executed as an Unix application, but it does not guarantee continuous media stream retrieval at constant rates. This means that the applications cannot play back high quality continuous media in a timely fashion.

In this paper, we describe the design and implementation of a simple continuous media storage system, CRAS, on Real-Time Mach[6, 7, 17] which has been developed by CMU, Keio University and JAIST. CRAS is more suitable for a wider variety of applications than traditional continuous media storage systems due to its use of a microkernel-based operating system. CRAS solves the problems described in the previous paragraphs by employing a real-time microkernel.

The following design decisions were made to keep CRAS small and compact:

- CRAS adopts the same disk layout policy as the Unix file system. Thus, both file systems access the same files, and functionality that does not require real-time constraints such as system administration is processed by the Unix file system.
- CRAS provides a single function, a constant rate retrieval for playback. This makes the size of CRAS compact. Also, it is easy to add a storage server that provides another function when necessary since the server can run on the Real-Time Mach microkernel as a user-level server.
- CRAS uses the real-time capabilities provided by Real-Time Mach to ensure constant rate retrievals. CRAS consists of several threads scheduled by the microkernel. This simplifies the structure of CRAS.

Real-Time Mach is currently integrated with the Lites server[3] which provides FreeBSD, NetBSD and Linux binary compatibility. Thus, CRAS on Real-Time Mach adds multimedia functionality to a traditional Unix environment.

In addition, We compare the throughput and delay jitter of CRAS with those of the Unix file system's using a movie player application, and we show that CRAS can not only meet the timing constraints of continuous media, but also provide high throughput even though it is compact and simple.

The remainder of this paper is structured as follows. Section 2 presents the design and implementation of

our continuous media storage server, CRAS on Real-Time Mach. Section 3 evaluates the performance of CRAS and section 4 summarizes the paper.

2 CRAS: A Constant Rate Access Server

The current version of CRAS is implemented as a server on Real-Time Mach. The following design goals were considered in the design of CRAS:

1. It should be compact and simple, and it should not require changing the Unix file system format.
2. It should retrieve multiple continuous media streams at constant rates.
3. It should provide a data transmission mechanism between CRAS and applications that support dynamic QOS (Quality Of Service) control.
4. It should provide a high-level application interface for continuous media applications.
5. It should be usable by various types of applications¹ beyond those that continuous media storage systems can support.

In this section, we describe how CRAS has met these design goals.

2.1 Overview of CRAS

Figure 1 shows our typical playback application on Real-Time Mach. The application accesses CRAS for constant rate data retrieval; this is the only functionality supported by CRAS.

The implementation of 'Fast Forward', 'Fast Rewind', 'Step by Frame', and similar operations uses the Unix file system because these operations do not require critical real-time response. CRAS and the Unix file system can share files because they use the same disk layout. This approach makes CRAS simple and allows it to be highly optimized for a single function: constant rate data stream retrieval. CRAS does not support retrieving data at a rate faster than the rate at which a stream was recorded since such a function is not necessary for most of applications.

One problem of the disk layout policy of the Unix file system is that the allocation of blocks in a single file may be random, making it difficult to provide a constant rate stream retrieval with high throughput. We minimized this problem by using the 'tunefs' command at file system creation time to indicate that

¹In our research, we focus on building personal interactive continuous media applications.

blocks should be allocated as contiguously as possible².

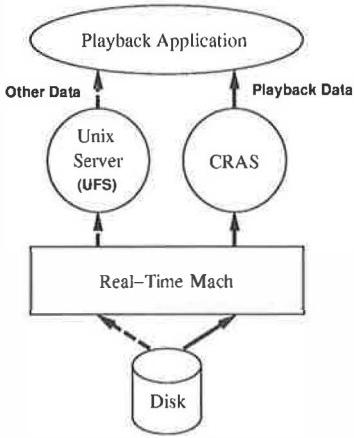


Figure 1: Architecture of a Playback Application on Real-Time Mach

A serious problem for continuous media servers is that they may trigger page faults that block threads in the servers, and interrupting a constant rate stream retrieval in progress. Our approach is to make the server small enough³ so it can wire down all memory allocated for the server without severely impacting the amount of memory available to other applications. Also, the server is carefully designed to avoid accessing any non real-time OS servers during constant rate retrieval, so the priority inversion problem can be prevented from occurring[7].

Figure 2 shows an overview of how CRAS works. An application sends a request to CRAS to open a continuous media session, and retrieve media data from a file. CRAS performs an admission test to determine whether sufficient capacity is available to perform the requested retrieval. If the admission test fails, the application needs to wait for the completion of currently running retrievals. After the admission test passes, CRAS creates a shared memory between CRAS and the application for delivering media data from CRAS to the application(1).

CRAS schedules read requests of all admitted sessions periodically, and sends them to the kernel's device driver to start the disk operations(2). Then, the driver starts the operations(3). When completion interrupts are received by the device driver(4), CRAS is notified. Then, CRAS puts the media data retrieved from disk into the shared buffer between the applications and CRAS(5).

²Some recent Unix Fast File Systems adopt the same policy to improve sequential read throughput of files.

³CRAS consumes about (250KB + total buffer space) of physical memory.

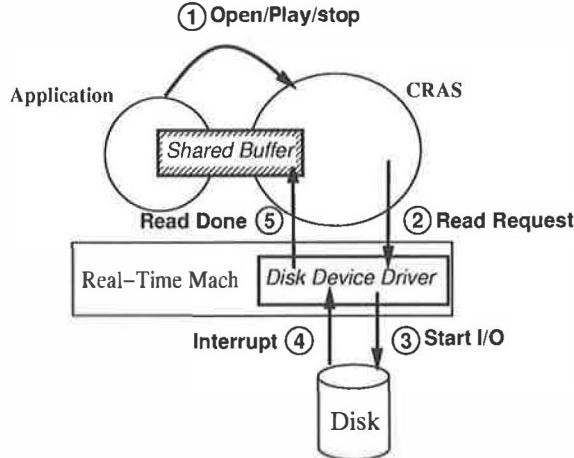


Figure 2: Interaction between an Application and CRAS

In Section 2.2, we describe a scheduling policy for guaranteeing constant rate retrievals. Section 2.3 presents an admission test used in CRAS. In Section 2.4, we describe the shared memory communication mechanism between CRAS and applications. Section 2.5 describes the application interface of CRAS. Finally, we describe how implementing CRAS on a microkernel enables it to be used in a variety of system configurations.

2.2 Retrieving Continuous Media at a Constant Rate

Continuous media data are usually accessed sequentially and it is easy to predict which data regions will be accessed in the future. In traditional storage systems, a read-ahead policy is used for improving sequential access performance, but it does not ensure that all data will arrive in memory by the time that it is needed. CRAS schedules pre-fetches of all data which are required in the near future and ensures that all required data will be fetched by the time it is needed.

CRAS periodically schedules pre-fetches of media data which are needed in the next period, and ensures that these data are fetched and placed in memory before the end of the current period. We call this period of CRAS its *interval time*. The interval time is determined by a tradeoff between the maximum number of streams supported by CRAS and the initial delay of the output streams. CRAS calculates the data sizes which are required in order to read media data within the interval time for the output streams. For example, CRAS retrieves all video frames that an application requires to play back within the interval time in a shared buffer, and the application fetches each video frame

from the shared buffer at the video stream's frame rate. CRAS optimizes throughput by reading continuous media data from disks up to 256K bytes at a time when the data is stored contiguously and by making all the read requests to disks in cylinder order to minimize the seek time. If the size of contiguous blocks is less than 256K bytes, CRAS reads the smaller blocks instead of reading a big block, decreasing the overall throughput.

Consider a video stream that was recorded at 30 fps. If an application wants to play back the video stream at 60 fps(Fast Forward), CRAS needs to retrieve all the video frames at twice the normal speed since CRAS cannot skip video frames during the retrieval. If the application will skip video frames during the retrieval and the retrieval does not require timeliness, the Unix file system should be used. As described in Section 2.4, an application can skip any video frames without disturbing the retrieval of CRAS although CRAS retrieves all video frames into the shared buffer.

Figure 3 shows the structure of CRAS. CRAS includes five threads: a request manager thread, a request scheduler thread, a deadline manager thread, an I/O done manager thread, and a signal handler thread. The request manager thread accepts open/close requests from applications and calculates the buffer size that must be read for each active stream within the interval time. The request scheduler thread sends read requests to the kernel for all active streams. At the first phase of each interval, the request scheduler thread gets all the data that has been retrieved in the previous interval from the I/O done queue. The request scheduler then puts this data and its associated timestamp in the shared memory buffer which is used to pass data to applications. This thread also requests all necessary disk reads for the next interval from the kernel. Now, the I/O done manager thread again accepts I/O done notifications from the kernel, and puts the media data into the I/O done queue. The deadline manager thread manages the deadline of the request scheduler thread and executes the recovery action from a missed deadline. Currently, CRAS notifies a warning message when a deadline is messed.

We changed two parts of Real-Time Mach to support CRAS. The first modification was to the queue in the disk driver. We divided the queue into two queues, one for normal activities, and another for real-time activities. CRAS uses the real-time queue and the Unix file system uses the non real-time queue. If there are any requests in the real-time queue, the requests are processed before the request in the non real-time queue. Each queue is sorted by using the traditional C-SCAN algorithm⁴ to minimize total seek time. The

second modification we made is add a new interface for reading larger data blocks from the kernel efficiently. Existing primitives for reading data from devices allocate buffers in kernel, making it difficult to manage buffers in CRAS explicitly. Thus, the new interface enables us to specify buffers that are managed in CRAS explicitly.

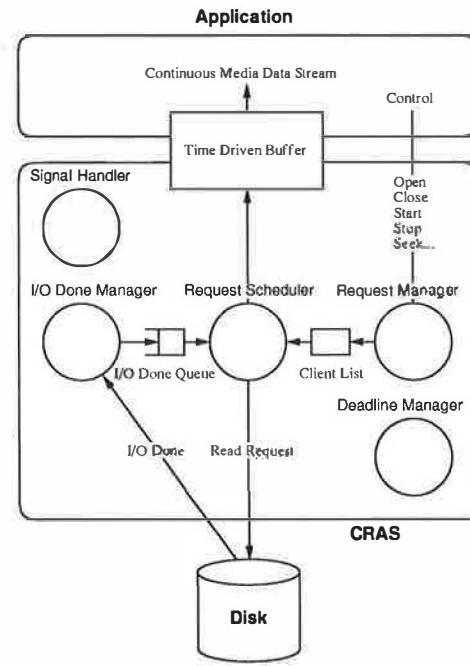


Figure 3: Structure of CRAS

2.3 Admission Test

When playing back continuous media, a host must not be interrupted by a resource shortage. Admission control is necessary to decide whether the real time requirements of a new continuous media stream can be satisfied before playback starts.

The admission test adopted in CRAS is based on estimating the time of data transfer. First, an application sends the data rate of a continuous media stream to CRAS when opening the file. Next, CRAS calculates the time for retrieving data [formula (1)] and the total size of buffer memory [formula (2)] that are required to handle the new stream based on the parameters of the disk (see Table 1) and the data rate of the new stream⁵. The data retrieval time is calculated as sum of two parts, data transfer time and overhead time. Overhead is sum of the time used by

⁴The disk arm moves unidirectionally across the disk surface toward the inner track. When there are no more requests for

⁵Appendix B describes the derivation of these formulas.

other disk access activities, head seek time, rotational delay and command overhead. Appendix C describes total overhead time, O_{total} , in more detail.

T	Interval time of CRAS
B_{total}	Total buffer memory size
D	Disk data transfer rate
O_{total}	Total access overhead
C_{total}	Total size of chunks
R_{total}	Total data rate

Table 1: Parameters for Admission Test

$$T \geq \frac{O_{total} \times D + C_{total}}{D - R_{total}} \quad (1)$$

$$B_{total} \geq 2 \times (T \times R_{total} + C_{total}) \quad (2)$$

If the time for retrieving data is less than the interval time and the total size of buffer memory (including the buffer for the new stream) is less than the total memory size allocated for CRAS, a new stream is opened successfully and CRAS allocates buffers and starts to pre-fetch the stream. Otherwise, the admission test fails and an error indicating a resource shortage is returned to the application.

2.4 Time-Driven Shared Memory Buffer

There is a problem when using traditional FIFO buffers for communicating between client applications and the continuous media server. Since CRAS delivers data to buffers at a constant rate, when applications cannot fetch data from the buffers at the same rate, the buffers may overflow. For this situation, FIFO buffers have the undesirable logical property of discarding incoming new data before obsolete old data in the buffers.

Our system uses a logical clock per stream to control retrieval; this clock is distinct from the system clock. The speed of a stream determines the rate of advance of the associated logical clock. At the time when a stream is opened, the logical clock is set to zero, and its rate of advance is set to the original recording data rate of the stream. CRAS schedules pre-fetcheds according to the logical rate, and clients access media data using a logical clock.

The shared memory between CRAS and applications described in the previous section is called a *time-driven shared memory buffer*. This buffer enables clients to access media data without explicit communication with CRAS. When a new stream is opened, a new shared memory area is set up for the stream.

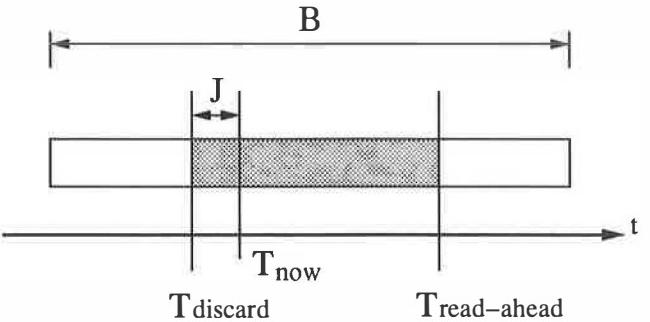


Figure 4: Time-Driven Shared Memory Buffer

CRAS puts the media data with its timestamp in the buffer. CRAS removes the media data automatically when the timestamp becomes greater than the logical clock's current time. Thus, the buffer always has enough space for storing media data retrieved from disks.

The merit of this approach is that it allows a client to change the rate at which it processes a media stream without changing the rate of retrieval in CRAS or stopping the current stream. The scheme is especially useful for building an application supporting dynamic QOS control[8]. The mechanism is difficult to implement using a traditional FIFO buffer, because new data is dropped when the buffer is full. *Time-driven shared memory buffer* solves the problem by discarding obsolete data from the buffer automatically based on timestamps of the data.

Figure 4 shows the structure of the time-driven shared memory buffer. In this figure, B stands for the total size of the buffers, and J is a short time which allows small jitters to occur. $T_{discard}$ defines the time when all data whose timestamp is less than this time to be discarded. T_{now} is the current time, and $T_{read-ahead}$ is the time when the next pre-fetches of media data will start. Here, $T_{discard}$ is defined as $T_{now} - J$. Clients read the data at the location pointed to by T_{now} . Time-driven shared memory buffer is attractive when two periodic threads are executing at different rates. Our scheme solves the problem by decoupling each period of a thread from other threads processing the same stream by using a time-driven shared memory buffer. The solution provides a better support for dynamic QOS control.

Consider how the time-driven shared memory buffer works when the rate of an application is different from the rate of a media stream. Let us assume a video stream with 30 fps, and the interval of CRAS is 1 second. CRAS retrieves 30 video frames for each 1 sec interval from a disk, and puts them into the shared

<code>crs_open</code>	Open a new continuous media stream
<code>crs_close</code>	Close a continuous media stream
<code>crs_start</code>	Start the logical clock of a continuous media stream
<code>crs_stop</code>	Stop the logical clock of a continuous media stream
<code>crs_seek</code>	Set the logical clock to the specified value
<code>crs_get</code>	Get the address of data chunk in the time-driven shared memory buffer specified by logical time

Table 2: Application interface of CRAS

buffer. Now, an application wants to play back the video stream at 10 fps. The application fetches video frames whose timestamps are equal to the logical clock that is updated every 100 ms. This means that the application only uses one of every three frames from the shared buffer. Then, the video frames are automatically removed from the buffer when their timestamps become greater than the logical clock. The scheme does not require a complex feedback mechanism when the rate of an application and a media stream are different.

2.5 Application Interface

The application interface of CRAS is different from conventional storage systems due to CRAS's timing-dependency. To ensure a constant rate stream retrieval, CRAS pre-fetches data blocks without explicit requests from applications. This requires a high-level application interface that is suitable for building continuous media applications, rather than a traditional file system interface.

CRAS does not provide an explicit read request for applications. In a traditional file system interface, pre-fetching cannot be controlled from users. However, pre-fetching is the most important mechanism for maintaining constant rate stream retrieval. Hence, the interface should support primitives to initiate pre-fetching to start a stream, and to stop pre-fetching in order to suspend the stream. A traditional interface does not offer such primitives so it is not suitable for continuous media storage systems.

The application interface of CRAS is shown in Table 2. In contrast to a conventional interface that retrieves data by specifying a byte offset in a file, the CRAS interface retrieves a chunk of data specified by a logical clock value. Since the interface is based on the media data's logical clock, an application program can fetch the necessary data blocks according to its actual requirements.

In Table 2, `crs_open`, `crs_close`, `crs_start`, `crs_stop` and `crs_seek` are requests to CRAS. Unlike these, `crs_get` does not communicate with CRAS, because an application can get the data from its time-driven

shared memory buffer. `Crss_open` is used to open a new continuous media stream, and `crs_close` closes the stream. `Crss_start` is used to start pre-fetching, and continuous media data is put at a constant rate into a time-driven shared memory buffer. `Crss_stop` stops the pre-fetching.

When an application opens a new continuous media stream by using `crs_open`, the application sends information about the timestamp, duration and size of each chunk to CRAS. The information is used for scheduling pre-fetched media data and discarding obsolete media data in CRAS. Usually, this timing information is stored in a control file separate from the continuous media data file or is calculated by the client application. The timestamp of each block, which is used for reading data from applications and for discarding obsolete data, is calculated from the sum of the durations of all previous media blocks.

2.6 Customization of CRAS

Since it is difficult to implement a specialized storage systems for each new continuous media application, we have designed CRAS to be used by various types of applications ranging from a large-scale video-on-demand system to a small scale independent server accessed by several personal applications on Unix. User-level implementation of a continuous media storage system allows us to customize the system easily, and allows the system to execute multiple CRAS's simultaneously.

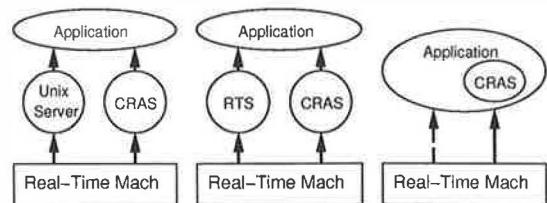


Figure 5: Customization of CRAS

Figure 5 shows several configurations of CRAS. The left configuration is the typical configuration where

a Unix Server being used. The middle configuration shows CRAS being used with RTS, a small operating system server specialized for embedded systems on Real-Time Mach[6]. The right configuration is more aggressive, because CRAS is linked with an application. This configuration is especially useful for supporting continuous media in embedded systems.

3 Evaluation and Application Experience

This section describes the evaluation of CRAS, and our experiences with using CRAS in a QuickTime player.

3.1 Evaluation of CRAS

The results in this section show the advantages of CRAS over the Unix file system and the importance of the real-time scheduling techniques in CRAS. We evaluated the basic performance of CRAS on a Gateway2000 P5-100 with a 100MHz Intel Pentium processor, 32MB of memory, 2GB of SCSI disk(Seagate ST32550N) whose data transfer bandwidth is about 6.5M bytes per second(Table 4 in Appendix A shows the actual disk parameters of the disk.), and a 10 Mbps Ethernet interface. We used a timer board which contains an AM9513 counter/timer module for measurements with accuracy to the nearest 1 micro second⁶.

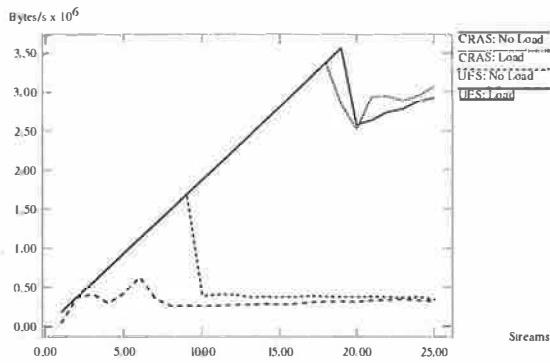


Figure 6: CRAS vs. UFS Throughput

Figure 6 shows the results of the first benchmark, demonstrating the advantages of CRAS over the Unix file system. In this benchmark, the data rate of each stream is 1.5Mbps⁷, and either CRAS or Unix file system is used to read several streams simultaneously. The benchmark is evaluated both with no disk I/O

⁶Real-Time Mach supports high resolution timers using this timer board.

⁷This rate corresponds to a MPEG1 data stream.

activity and with other disk I/O activity⁸. The number of streams is varied from 1 to 25, and the actual throughput that CRAS and the Unix file system can achieve is measured in each case. In this benchmark, the interval time of CRAS was 0.5 second, and the initial delay time of each stream was 1 second. The results show that CRAS can support more streams, and provide a higher throughput than the Unix file system can achieve, because the Unix file system may cause priority inversions.

The result also shows that CRAS can achieve 55% of the disk's maximum transfer rate, and that background file access activities do not affect the throughput of CRAS. If a longer initial delay is allowed, CRAS can support more streams or higher data rates. For example, with 3 seconds initial delay, it can support more than 25 MPEG1 streams whose total throughput is 4.6MB/s(70% of disk bandwidth).

On the other hand, the Unix file system provides up to nine streams without other disk I/O traffic. Moreover, it cannot support even one stream when other disk I/O traffic is present.

Figure 7 is the result of the second benchmark, showing another advantage of CRAS over the Unix file system. In this benchmark, an application retrieves a video stream from each of the file systems, and we measured each frame's delay (the difference between current time and logical time) while running other activities that access the same disk. The result shows that the Unix file system causes larger delay jitters of video frames than CRAS even when both file systems achieve the same throughput.

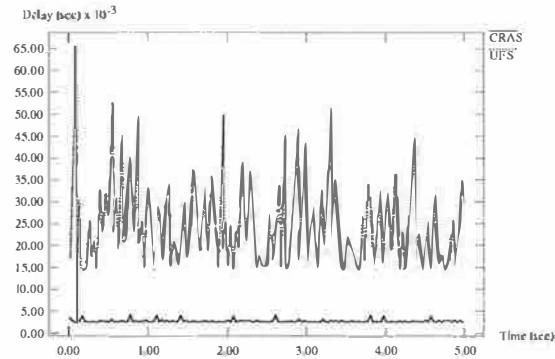


Figure 7: CRAS vs. UFS Delay

Figure 8 and 9 show the accuracy of our admission test. In the benchmarks, we employ two data

⁸We executed two 'cat' programs which read movie files with the benchmark program. The priority of the benchmark program is higher than the priorities of 'cat' programs.

rates 1.5Mbps and 6Mbps⁹, and we vary the number of streams from 1 to 20 for 1.5Mbps streams and from 1 to 5 for 6Mbps streams. Then, we measured the average and maximum ratio of the actual disk I/O time to the calculated I/O time, where 100% means that the CRAS's estimation of disk I/O time is perfect, and a lower ratio means that the estimation is more pessimistic. Further, we ran some disk I/O intensive applications together with CRAS. In Figure 8 and 9, “no_load” indicates that only CRAS was running, and “load” indicates that CRAS and these applications were simultaneously running.

The results show that the estimations of the admission test are very pessimistic when there are few streams, especially when the data rates of streams are low. This is because that the admission tests uses worst case seek time and rotational delay time to estimate total disk access time. If there are few streams or the data rates of streams are low, the time which takes to transfer data is short and overhead time dominates the calculated cost. On the other hand, the cases of 6Mbps streams with background activities yield an accuracy to about 70%. These results show that our admission control is over estimated when the data rate of each stream is low or small number of sessions are opened. If the data rate of a session is increased, the data transfer time will dominate the more pessimistic overhead estimates lessening the pessimism of our admission test.

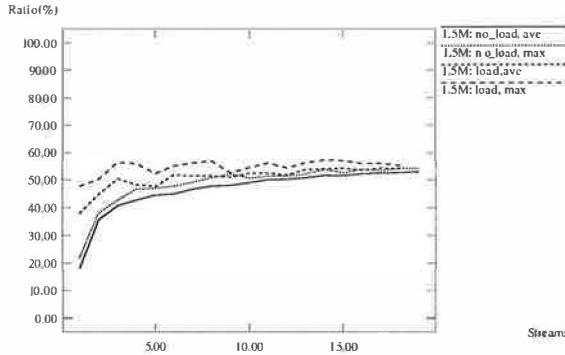


Figure 8: Accuracy of Admission Test (1): 1.5Mbps

Figure 10 shows the importance of real-time scheduling techniques to guarantee constant rate stream retrieval. In this benchmark, one stream of 1.5Mbps was retrieved from CRAS while other tasks that consume CPU time were also running. We measured each frame's delay under both fixed priority scheduling and round-robin scheduling. Under round-robin scheduling, delay jitters of retrieved data are much larger than

⁹The data rate corresponds to MPEG2 data stream.

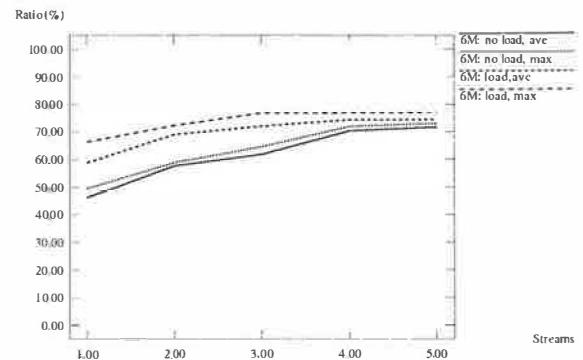


Figure 9: Accuracy of Admission Test (2): 6Mbps

under fixed priority scheduling. This result shows that real-time scheduling is very important to retrieve continuous media data at a constant rate.

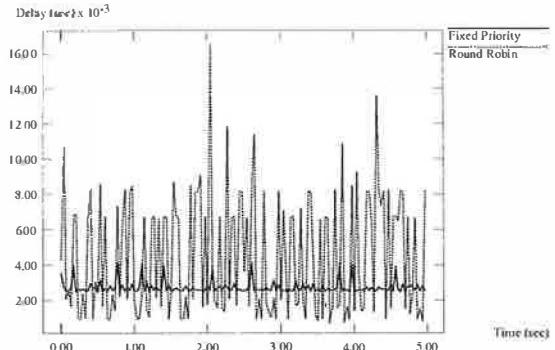


Figure 10: Effect of Real-Time scheduling

3.2 Experience and Discussion

We implemented a distributed QuickTime movie player, QtPlay[8, 16] which retrieves movie data from disks using CRAS and transmits it over the network using NPS[9]. QtPlay sends video streams to the X11 server and audio streams to an audio server for playback, and it can play multiple movies simultaneously. Our experience with implementing QtPlay using CRAS shows that CRAS's high level application interface is suitable for handling continuous media, and makes it easy to construct continuous media applications. This application demonstrates that CRAS can support constant rate stream retrievals even though it is very compact and simple, and that it is particularly suitable for playback applications.

Our QuickTime player can change the frame rate of

a movie at any time without notifying CRAS because CRAS's time-driven shared buffer enables applications to support this flexibility. This ability is very attractive for personal or small scale distributed environments because applications cannot be executed at a full quality due to the limited hardware capabilities of the machines used in these environments.

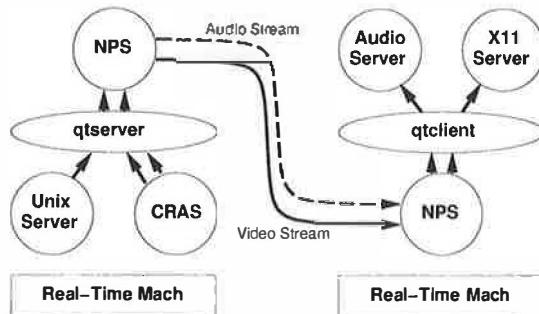


Figure 11: QuickTime Player on Real-Time Mach

We found three problems with CRAS in the personal environment. First, the sizes of video data compressed by JPEG or MPEG varies significantly. In this case, the rate of a stream is not constant. CRAS allocates buffers for retrieving within each interval time based on worst case bandwidth. If the average bandwidth is much less than the worst case bandwidth, much of the buffer space may not be used.

Our admission test algorithm causes the second problem. The result of the previous section shows that our admission test is more pessimistic when it is used by one or two sessions with low data rates. However, the rest of the throughput may be used by non real-time disk accesses.

Lastly, we adopt the Unix file system's disk layout policy with the changed parameter for allocating blocks contiguously as much as possible by 'tunefs' command. However, editing a continuous media file may make the layout of blocks random. Non-continuous data makes the seek time long, and the throughput of the disk is decreased. For example, adding a block in the middle of two contiguous blocks prevents CRAS from retrieving media data at a constant rate. Our approach needs to rearrange media files whose data blocks are allocated randomly.

4 Conclusions

In this paper, we presented the design and implementation of a continuous media storage server, CRAS, and showed that CRAS can retrieve media data without violating the timing constraints even though it is compact and simple.

CRAS provides mechanisms to support continuous media including a constant rate stream retrieval, an admission control mechanism, a high-level application interface, and a stream transmission mechanism for applications supporting dynamic QOS control. Also, CRAS demonstrates that a microkernel-based operating system is suitable for meeting the requirements of various types of applications in continuous media storage systems.

Although the current version of CRAS has no capability for writing continuous media files at constant rates, it is easy to add it. To limit the size of these modifications, the Unix file system must be modified to allocate data blocks in advance when a file is created or expanded. CRAS can then write continuous media data at constant rates to the allocated blocks via the same algorithm used for retrieving continuous media data.

Acknowledgements

We would like to thank the members of the MMMC Project in JAIST for their valuable comments and inputs to the development of Real-Time Mach and also for integrating Lites with Real-Time Mach. We are also grateful to David Black, the Usenix shepherd for our paper, who has provided many helpful comments.

References

- [1] D.A.Anderson, et. al. "A File System for Continuous Media", ACM Transaction on Computer Systems, Vol.10, No.4, 1992.
- [2] H.M.Deitel, "An Introduction to Operating Systems", Addison-Wesley, 1984.
- [3] J.Helander, "Unix under Mach: The Lites Server", Helsinki University of Technology, Master's Thesis, 1994.
- [4] E.M.Hoffer, et. al., "QuickTime: An Extensible Standard for Digital Multimedia", In Proceedings of IEEE COMPON, 92, 1992.
- [5] P.Lougher, D.Sherpherd, "The Design of a Storage Server for Continuous Media", The Computer Journal, Vol.36, No.1, 1992, pp.32-42.
- [6] T.Nakajima, T.Kitayama and H.Tokuda, "Experiments with Real-Time Servers in Real-Time Mach", In Proceedings of the USENIX 3rd Mach Symposium, 1993.
- [7] T.Nakajima, T.Kitayama, H.Arakawa, and H.Tokuda, "Integrated Management of Priority Inversion in Real-Time Mach", In Proceedings of the Real-Time System Symposium, 1993.

- [8] T.Nakajima and H.Tezuka, "A Continuous Media Application supporting Dynamic QOS Control on Real-Time Mach", In Proceedings of the ACM Multimedia '94, 1994.
- [9] T.Nakajima, "NPS:User-Level Real-Time Network Engine on Real-Time Mach", In Proceedings of First International Workshop on Real-Time Computing System and Applications, 1994.
- [10] T.Nakajima and H.Tokuda, "Design and Implementation of a User-Level Real-Time Network Engine", IS-RR-94-14S, Research Report, Japan Advanced Institute of Science and Technology, 1994.
- [11] S.Ramanathan, H.M. Vin, and P.V.Rangan, "Towards Personalized Multimedia Dial-Up Services", Computer Networks and ISDN Systems, 1994.
- [12] K.K.Ramakrishnan, L.Vaitzblit, C.Gray, U.Vahalia, D.Ting, P.Tzelnic, S.Glaser, W.Duso, "Operating System Support for a Video-on-Demand File Service", Multimedia Systems, Vol.3, No.2, Springer-Verlag, 1995.
- [13] P.V. Rangan, Harrick M.Vin, "Efficient Storage Techniques for Digital Continuous Multimedia", IEEE Transactions on Knowledge and Data Engineering, August, 1993.
- [14] L.A.Rowe and B.C.Smith, "A Continuous Media Player", In Proceedings of Third International Workshop on Network and Operating System Support for Digital Audio and Video, November, 1992.
- [15] C.Ruemmler and J.Wilkes, "An Introduction to Disk Drive Modelling", IEEE Computer vol.27, num.3, pp.17-28, March 1994.
- [16] H.Tezuka and T.Nakajima, "Experiences with building a Continuous Media Application on Real-Time Mach", In Proceedings of Second International Workshop on Real-Time Computing Systems and Applications, 1995.
- [17] H.Tokuda, T.Nakajima, and P.Rao, "Real-Time Mach: Towards a Predictable Real-Time System", In Proceeding of the USENIX 1st Mach Symposium, October, 1990.
- [18] Microsoft Windows, "Multimedia: Programmer's Workbook", Microsoft Press, 1991.
- [19] H.M.Vin, P.V.Rangan, "Designing a Multi-User HDTV Storage Server", IEEE Journal on Selected Areas in Communication, Vol.11, No.1, January, 1993.

A Disk Parameters

Parameters used in the admission test of CRAS are summarized in Table 3. In the table, T_{rot} is calculated by a disk's spindle rotation speed, and D , T_{seek_max} , T_{seek_min} and T_{cmd} are measured using small benchmark programs.

N	Number of continuous media stream
T	Interval time of CRAS
B_{total}	Total buffer memory size
D	Disk data transfer rate
T_{seek_max}	Maximum head seek time
T_{seek_min}	Minimum head seek time
T_{rot}	Disk rotational latency
T_{cmd}	Disk command overhead
B_{other}	Maximum block size of other disk traffic
O_{total}	Total access overhead
C_{total}	Total size of chunks
R_{total}	Total data rate
O_{other}	Delay time by other disk traffics
O_{seek}	Total head seek time
O_{rot}	Total disk rotational delay
O_{cmd}	Total disk command overhead

Table 3: Parameters for Admission Test

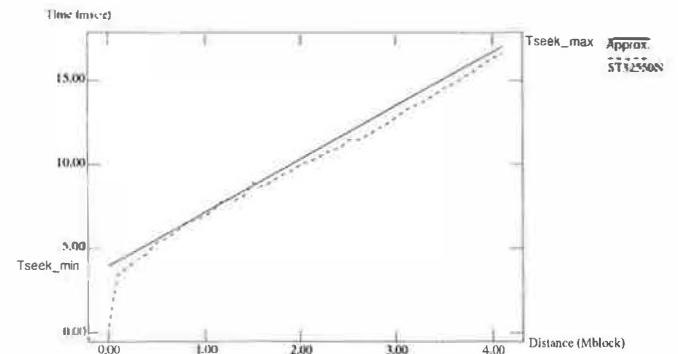


Figure 12: Disk Seek Time

Figure 12 shows the measured seek time of the disk that we used. T_{seek_max} and T_{seek_min} are obtained by approximating the measured results linearly. Table 4 shows the measured parameters of the disk.

B Admission Test of CRAS

In this section, we describe how to derive formulas (1) and (2) presented in Section 2.3.

D	6.5MB/s
T_{seek_max}	17ms
T_{seek_min}	4ms
T_{rot}	8.33ms
T_{cmd}	2ms
B_{other}	64KB

Table 4: Actual Disk Parameters of our system

We indicate the parameters of each data stream with suffix i . A_i denotes the amount of data that is retrieved for each stream within an interval time T . A_i is derived as follows.

$$A_i \geq T \times R_i + C_i \quad (3)$$

Also, interval time T should satisfy the following formula where $O_i + A_i/D$ indicates time which requires to retrieve media data for session i within the interval.

$$T \geq \sum_{i=1}^N (O_i + \frac{A_i}{D}) \quad (4)$$

Substituting formula (3) into (4), the interval time T of CRAS must satisfy the next formula.

$$T \geq \frac{O_{total} \times D + C_{total}}{D - R_{total}} \quad (5)$$

$$O_{total} = \sum_{i=1}^N O_i, \quad C_{total} = \sum_{i=1}^N C_i, \quad R_{total} = \sum_{i=1}^N R_i$$

Then, the total amount of data A_{total} which must be retrieved within interval time T , buffer size of each stream B_i and total amount of buffer B_{total} are calculated in the following way.

$$\begin{aligned} A_{total} &= \sum_{i=1}^N A_i \\ &= T \times R_{total} + C_{total} \end{aligned} \quad (6)$$

$$\begin{aligned} B_i &= 2 \times A_i \\ &= 2 \times (T \times R_i + C_i) \end{aligned} \quad (7)$$

$$\begin{aligned} B_{total} &= \sum_{i=1}^N B_i \\ &= 2 \times (T \times R_{total} + C_{total}) \end{aligned} \quad (8)$$

C Calculating Disk Access Overhead

In this appendix, we describe the calculation of the disk access overheads that our admission test uses in details. The disk access overhead is calculated as the sum of an overhead of other disk access activities(C.1) a command overhead(C.2), a head seek time(C.3), and a disk rotational delay(C.4).

C.1 Overhead of Other Disk Access Activities: O_{other}

CRAS cannot access a disk when another activity is accessing the disk since its disk access operations are not interrupted. Thus, the time which is consumed by the activity is taken into account as an overhead that delays the disk accesses by CRAS. The worst case overhead is calculated as follows.

$$O_{other} = T_{cmd} + T_{seek_max} + T_{rot} + \frac{B_{other}}{D} \quad (9)$$

C.2 Command Overhead: O_{cmd}

Command overhead T_{cmd} is the time which is necessary to setup each disk access operation. When N reads are performed, total command overhead O_{cmd} is as follows.

$$O_{cmd} = N \times T_{cmd} \quad (10)$$

C.3 Head Seek Time: O_{seek}

Since the seek time is not proportional to cylinder distance[15], it is difficult to calculate the seek time from one cylinder to another cylinder. Thus, we use the linear approximation to estimate seek time T_{seek} in the following way.

$$\begin{aligned} T_{seek}(x) &= \alpha \times x + \beta \\ \alpha &= \frac{T_{seek_max} - T_{seek_min}}{N_{cyl}} \\ \beta &= T_{seek_min} \end{aligned}$$

In the above formula, x is a cylinder distance and N_{cyl} is the total number of cylinders of the disk.

Since CRAS sorts disk access requests in cylinder order, we obtain the following formula for estimating the total seek time to access $N(\geq 2)$ streams, where $d_{i,j}$ indicates the cylinder distance of stream i and j .

$$O_{seek} = T_{seek_max} + \sum_{i=1}^{N-1} T_{seek}(d_{i,i+1})$$

$$= T_{\text{seek_max}} + \alpha \times d_{1,N} + (N - 1) \times \beta$$

Assuming the worst case, $d_{1,N}$ is equal to N_{cyl} . We obtain O_{seek} as follows.

$$O_{\text{seek}}(1) = T_{\text{seek_max}} \quad (11)$$

$$\begin{aligned} O_{\text{seek}}(N) &= T_{\text{seek_max}} + \alpha \times N_{\text{cyl}} + (N - 1) \times \beta \\ &= 2 \times T_{\text{seek_max}} \\ &\quad + (N - 2) \times T_{\text{seek_min}} \end{aligned} \quad (12)$$

In formula (12), $T_{\text{seek_max}}$ indicates the worst case seek time for moving heads to the outer track. The seek is required to take into account non real-time disk accesses. $T_{\text{seek_max}} + (N - 2) \times T_{\text{seek_min}}$ indicates the total seek time for processing all requests that are sorted in cylinder order within an interval.

C.4 Disk Rotational Delay: O_{rot}

A maximum rotational delay for accessing an arbitrary data block is equal to the disk rotation time T_{rot} . Thus, total disk rotational delay O_{rot} is calculated as follows.

$$O_{\text{rot}} = N \times T_{\text{rot}} \quad (13)$$

C.5 Total Overhead: O_{total}

From formulas (9), (10), (11), (12) and (13), we can calculate the total overhead time O_{total} for accessing 1 stream in formula (14) and N (≥ 2) streams in (15).

$$\begin{aligned} O_{\text{total}} &= O_{\text{other}} + O_{\text{seek}} + O_{\text{rot}} + O_{\text{cmd}} \\ O_{\text{total}}(1) &= \frac{B_{\text{other}}}{D} + 2 \times (T_{\text{seek_max}} + T_{\text{rot}} + T_{\text{cmd}}) \end{aligned} \quad (14)$$

$$\begin{aligned} O_{\text{total}}(N) &= \frac{B_{\text{other}}}{D} + 3 \times T_{\text{seek_max}} + (N - 2) \times T_{\text{seek_min}} \\ &\quad + (N + 1) \times (T_{\text{rot}} + T_{\text{cmd}}) \end{aligned} \quad (15)$$

Eliminating Receive Livelock in an Interrupt-driven Kernel

Jeffrey C. Mogul

Digital Equipment Corporation Western Research Laboratory

K. K. Ramakrishnan

AT&T Bell Laboratories

Abstract

Most operating systems use interface interrupts to schedule network tasks. Interrupt-driven systems can provide low overhead and good latency at low offered load, but degrade significantly at higher arrival rates unless care is taken to prevent several pathologies. These are various forms of *receive livelock*, in which the system spends all its time processing interrupts, to the exclusion of other necessary tasks. Under extreme conditions, no packets are delivered to the user application or the output of the system.

To avoid livelock and related problems, an operating system must schedule network interrupt handling as carefully as it schedules process execution. We modified an interrupt-driven networking implementation to do so; this eliminates receive livelock without degrading other aspects of system performance. We present measurements demonstrating the success of our approach.

1. Introduction

Most operating systems use interrupts to internally schedule the performance of tasks related to I/O events, and particularly the invocation of network protocol software. Interrupts are useful because they allow the CPU to spend most of its time doing useful processing, yet respond quickly to events without constantly having to poll for event arrivals.

Polling is expensive, especially when I/O events are relatively rare, as is the case with disks, which seldom interrupt more than a few hundred times per second. Polling can also increase the latency of response to an event. Modern systems can respond to an interrupt in a few tens of microseconds; to achieve the same latency using polling, the system would have to poll tens of thousands of times per second, which would create excessive overhead. For a general-purpose system, an interrupt-driven design works best.

Most extant operating systems were designed to handle I/O devices that interrupt every few milliseconds. Disks tended to issue events on the order

of once per revolution; first-generation LAN environments tend to generate a few hundred packets per second for any single end-system. Although people understood the need to reduce the cost of taking an interrupt, in general this cost was low enough that any normal system would spend only a fraction of its CPU time handling interrupts.

The world has changed. Operating systems typically use the same interrupt mechanisms to control both network processing and traditional I/O devices, yet many new applications can generate packets several orders of magnitude more often than a disk can generate seeks. Multimedia and other real-time applications will become widespread. Client-server applications, such as NFS, running on fast clients and servers can generate heavy RPC loads. Multicast and broadcast protocols subject innocent-bystander hosts to loads that do not interest them at all. As a result, network implementations must now deal with significantly higher event rates.

Many multi-media and client-server applications share another unpleasant property: unlike traditional network applications (Telnet, FTP, electronic mail), they are not flow-controlled. Some multi-media applications want constant-rate, low-latency service; RPC-based client-server applications often use datagram-style transports, instead of reliable, flow-controlled protocols. Note that whereas I/O devices such as disks generate interrupts only as a result of requests from the operating system, and so are inherently flow-controlled, network interfaces generate unsolicited receive interrupts.

The shift to higher event rates and non-flow-controlled protocols can subject a host to congestive collapse: once the event rate saturates the system, without a negative feedback loop to control the sources, there is no way to gracefully shed load. If the host runs at full throughput under these conditions, and gives fair service to all sources, this at least preserves the possibility of stability. But if throughput decreases as the offered load increases, the overall system becomes unstable.

Interrupt-driven systems tend to perform badly under overload. Tasks performed at interrupt level,

by definition, have absolute priority over all other tasks. If the event rate is high enough to cause the system to spend all of its time responding to interrupts, then nothing else will happen, and the system throughput will drop to zero. We call this condition *receive livelock*: the system is not deadlocked, but it makes no progress on any of its tasks.

Any purely interrupt-driven system using fixed interrupt priorities will suffer from receive livelock under input overload conditions. Once the input rate exceeds the reciprocal of the CPU cost of processing one input event, any task scheduled at a lower priority will not get a chance to run.

Yet we do not want to lightly discard the obvious benefits of an interrupt-driven design. Instead, we should integrate control of the network interrupt handling sub-system into the operating system's scheduling mechanisms and policies. In this paper, we present a number of simple modifications to the purely interrupt-driven model, and show that they guarantee throughput and improve latency under overload, while preserving the desirable qualities of an interrupt-driven system under light load.

2. Motivating applications

We were led to our investigations by a number of specific applications that can suffer from livelock. Such applications could be built on dedicated single-purpose systems, but are often built using a general-purpose system such as UNIX®, and we wanted to find a general solution to the livelock problem. The applications include:

- *Host-based routing*: Although inter-network routing is traditionally done using special-purpose (usually non-interrupt-driven) router systems, routing is often done using more conventional hosts. Virtually all Internet "firewall" products use UNIX or Windows NT™ systems for routing [7, 13]. Much experimentation with new routing algorithms is done on UNIX [2], especially for IP multicasting.
- *Passive network monitoring*: network managers, developers, and researchers commonly use UNIX systems, with their network interfaces in "promiscuous mode," to monitor traffic on a LAN for debugging or statistics gathering [8].
- *Network file service*: servers for protocols such as NFS are commonly built from UNIX systems.

These applications (and others like them, such as Web servers) are all potentially exposed to heavy, non-flow-controlled loads. We have encountered livelock in all three of these applications, have solved or mitigated the problem, and have shipped the solu-

tions to customers. The rest of this paper concentrates on host-based routing, since this simplifies the context of the problem and allows easy performance measurement.

3. Requirements for scheduling network tasks

Performance problems generally arise when a system is subjected to transient or long-term input overload. Ideally, the communication subsystem could handle the worst-case input load without saturating, but cost considerations often prevent us from building such powerful systems. Systems are usually sized to support a specified design-center load, and under overload the best we can ask for is controlled and graceful degradation.

When an end-system is involved in processing considerable network traffic, its performance depends critically on how its tasks are scheduled. The mechanisms and policies that schedule packet processing and other tasks should guarantee acceptable system *throughput*, reasonable *latency* and *jitter* (variance in delay), *fair* allocation of resources, and overall system *stability*, without imposing excessive overheads, especially when the system is overloaded.

We can define throughput as the rate at which the system delivers packets to their ultimate consumers. A consumer could be an application running on the receiving host, or the host could be acting as a router and forwarding packets to consumers on other hosts. We expect the throughput of a well-designed system to keep up with the offered load up to a point called the *Maximum Loss Free Receive Rate* (MLFRR), and at higher loads throughput should not drop below this rate.

Of course, useful throughput depends not just on successful reception of packets; the system must also transmit packets. Because packet reception and packet transmission often compete for the same resources, under input overload conditions the scheduling subsystem must ensure that packet transmission continues at an adequate rate.

Many applications, such as distributed systems and interactive multimedia, often depend more on low-latency, low-jitter communications than on high throughput. Even during overload, we want to avoid long queues, which increases latency, and bursty scheduling, which increases jitter.

When a host is overloaded with incoming network packets, it must also continue to process other tasks, so as to keep the system responsive to management and control requests, and to allow applications to make use of the arriving packets. The scheduling subsystem must fairly allocate CPU resources among packet reception, packet transmission, protocol

processing, other I/O processing, system housekeeping, and application processing.

A host that behaves badly when overloaded can also harm other systems on the network. Livelock in a router, for example, may cause the loss of control messages, or delay their processing. This can lead other routers to incorrectly infer link failure, causing incorrect routing information to propagate over the entire wide-area network. Worse, loss or delay of control messages can lead to network instability, by causing positive feedback in the generation of control traffic [10].

4. Interrupt-driven scheduling and its consequences

Scheduling policies and mechanisms significantly affect the throughput and latency of a system under overload. In an interrupt-driven operating system, the interrupt subsystem must be viewed as a component of the scheduling system, since it has a major role in determining what code runs when. We have observed that interrupt-driven systems have trouble meeting the requirements discussed in section 3.

In this section, we first describe the characteristics of an interrupt-driven system, and then identify three kinds of problems caused by network input overload in interrupt-driven systems:

- *Receive livelocks* under overload: delivered throughput drops to zero while the input overload persists.
- Increased *latency* for packet delivery or forwarding: the system delays the delivery of one packet while it processes the interrupts for subsequent packets, possibly of a burst.
- *Starvation* of packet transmission: even if the CPU keeps up with the input load, strict priority assignments may prevent it from transmitting any packets.

4.1. Description of an interrupt-driven system

An interrupt-driven system performs badly under network input overload because of the way in which it prioritizes the tasks executed as the result of network input. We begin by describing a typical operating system's structure for processing and prioritizing network tasks. We use the 4.2BSD [5] model for our example, but we have observed that other operating systems, such as VMS™, DOS, and Windows NT, and even several Ethernet chips, have similar characteristics and hence similar problems.

When a packet arrives, the network interface signals this event by interrupting the CPU. Device interrupts normally have a fixed Interrupt Priority Level (IPL), and preempt all tasks running at a lower

IPL; interrupts do not preempt tasks running at the same IPL. The interrupt causes entry into the associated network device driver, which does some initial processing of the packet. In 4.2BSD, only buffer management and data-link layer processing happens at "device IPL." The device driver then places the packet on a queue, and generates a software interrupt to cause further processing of the packet. The software interrupt is taken at a lower IPL, and so this protocol processing can be preempted by subsequent interrupts. (We avoid lengthy periods at high IPL, to reduce latency for handling certain other events.)

The queues between steps executed at different IPLs provide some insulation against packet losses due to transient overloads, but typically they have fixed length limits. When a packet should be queued but the queue is full, the system must drop the packet. The selection of proper queue limits, and thus the allocation of buffering among layers in the system, is critical to good performance, but beyond the scope of this paper.

Note that the operating system's scheduler does not participate in any of this activity, and in fact is entirely ignorant of it.

As a consequence of this structure, a heavy load of incoming packets could generate a high rate of interrupts at device IPL. Dispatching an interrupt is a costly operation, so to avoid this overhead, the network device driver attempts to *batch* interrupts. That is, if packets arrive in a burst, the interrupt handler attempts to process as many packets as possible before returning from the interrupt. This amortizes the cost of processing an interrupt over several packets.

Even with batching, a system overloaded with input packets will spend most of its time in the code that runs at device IPL. That is, the design gives absolute priority to processing incoming packets. At the time that 4.2BSD was developed, in the early 1980s, the rationale for this was that network adapters had little buffer memory, and so if the system failed to move a received packet promptly into main memory, a subsequent packet might be lost. (This is still a problem with low-cost interfaces.) Thus, systems derived from 4.2BSD do minimal processing at device IPL, and give this processing priority over all other network tasks.

Modern network adapters can receive many back-to-back packets without host intervention, either through the use of copious buffering or highly autonomous DMA engines. This insulates the system from the network, and eliminates much of the rationale for giving absolute priority to the first few steps of processing a received packet.

4.2. Receive livelock

In an interrupt-driven system, receiver interrupts take priority over all other activity. If packets arrive too fast, the system will spend all of its time processing receiver interrupts. It will therefore have no resources left to support delivery of the arriving packets to applications (or, in the case of a router, to forwarding and transmitting these packets). The useful throughput of the system will drop to zero.

Following [11], we refer to this condition as *receive livelock*: a state of the system where no useful progress is being made, because some necessary resource is entirely consumed with processing receiver interrupts. When the input load drops sufficiently, the system leaves this state, and is again able to make forward progress. This is not a deadlock state, from which the system would not recover even when the input rate drops to zero.

A system could behave in one of three ways as the input load increases. In an ideal system, the delivered throughput always matches the offered load. In a realizable system, the delivered throughput keeps up with the offered load up to the *Maximum Loss Free Receive Rate* (MLFRR), and then is relatively constant after that. At loads above the MLFRR, the system is still making progress, but it is dropping some of the offered input; typically, packets are dropped at a queue between processing steps that occur at different priorities.

In a system prone to receive livelock, however, throughput decreases with increasing offered load, for input rates above the MLFRR. Receive livelock occurs at the point where the throughput falls to zero. A livelocked system wastes all of the effort it puts into partially processing received packets, since they are all discarded.

Receiver-interrupt batching complicates the situation slightly. By improving system efficiency under heavy load, batching can increase the MLFRR. Batching can shift the livelock point but cannot, by itself, prevent livelock.

In section 6.2, we present measurements showing how livelock occurs in a practical situation. Additional measurements, and a more detailed discussion of the problem, are given in [11].

4.3. Receive latency under overload

Although interrupt-driven designs are normally thought of as a way to reduce latency, they can actually increase the latency of packet delivery. If a burst of packets arrives too rapidly, the system will do link-level processing of the entire burst before doing any higher-layer processing of the first packet, because link-level processing is done at a higher

priority. As a result, the first packet of the burst is not delivered to the user until link-level processing has been completed for all the packets in the burst. The latency to deliver the first packet in a burst is increased almost by the time it takes to receive the entire burst. If the burst is made up of several independent NFS RPC requests, for example, this means that the server's disk sits idle when it could be doing useful work.

One of the authors has previously described experiments demonstrating this effect [12].

4.4. Starvation of transmits under overload

In most systems, the packet transmission process consists of selecting packets from an output queue, handing them to the interface, waiting until the interface has sent the packet, and then releasing the associated buffer.

Packet transmission is often done at a lower priority than packet reception. This policy is superficially sound, because it minimizes the probability of packet loss when a burst of arriving packets exceeds the available buffer space. Reasonable operation of higher level protocols and applications, however, requires that transmit processing makes sufficient progress.

When the system is overloaded for long periods, use of a fixed lower priority for transmission leads to reduced throughput, or even complete cessation of packet transmission. Packets may be awaiting transmission, but the transmitting interface is idle. We call this *transmit starvation*.

Transmit starvation may occur if the transmitter interrupts at a lower priority than the receiver; or if they interrupt at the same priority, but the receiver's events are processed first by the driver; or if transmission completions are detected by polling, and the polling is done at a lower priority than receiver event processing.

This effect has also been described previously [12].

5. Avoiding livelock through better scheduling

In this section, we discuss several techniques to avoid receive livelocks. The techniques we discuss in this section include mechanisms to control the rate of incoming interrupts, polling-based mechanisms to ensure fair allocation of resources, and techniques to avoid unnecessary preemption.

5.1. Limiting the interrupt arrival rate

We can avoid or defer receive livelock by limiting the rate at which interrupts are imposed on the system. The system checks to see if interrupt processing is taking more than its share of resources, and if so, disables interrupts temporarily.

The system may infer impending livelock because it is discarding packets due to queue overflow, or because high-layer protocol processing or user code are making no progress, or by measuring the fraction of CPU cycles used for packet processing. Once the system has invested enough work in an incoming packet to the point where it is about to be queued, it makes more sense to process that packet to completion than to drop it and rescue a subsequently-arriving packet from being dropped at the receiving interface, a cycle that could repeat *ad infinitum*.

When the system is about to drop a received packet because an internal queue is full, this strongly suggests that it should disable input interrupts. The host can then make progress on the packets already queued for higher-level processing, which has the side-effect of freeing buffers to use for subsequent received packets. Meanwhile, if the receiving interface has sufficient buffering of its own, additional incoming packets may accumulate there for a while.

We also need a trigger for re-enabling input interrupts, to prevent unnecessary packet loss. Interrupts may be re-enabled when internal buffer space becomes available, or upon expiration of a timer.

We may also want the system to guarantee some progress for user-level code. The system can observe that, over some interval, it has spent too much time processing packet input and output events, and temporarily disable interrupts to give higher protocol layers and user processes time to run. On a processor with a fine-grained clock register, the packet-input code can record the clock value on entry, subtract that from the clock value seen on exit, and keep a sum of the deltas. If this sum (or a running average) exceeds a specified fraction of the total elapsed time, the kernel disables input interrupts. (Digital's GIGAswitch™ system uses a similar mechanism [15].)

On a system without a fine-grained clock, one can crudely simulate this approach by sampling the CPU state on every clock interrupt (clock interrupts typically preempt device interrupt processing). If the system finds itself in the midst of processing interrupts for a series of such samples, it can disable interrupts for a few clock ticks.

5.2. Use of polling

Limiting the interrupt rate prevents system saturation but might not guarantee progress; the system must also fairly allocate packet-handling resources between input and output processing, and between multiple interfaces. We can provide fairness by carefully polling all sources of packet events, using a round-robin schedule.

In a pure polling system, the scheduler would invoke the device driver to “listen” for incoming packets and for transmit completion events. This would control the amount of device-level processing, and could also fairly allocate resources among event sources, thus avoiding livelock. Simply polling at fixed intervals, however, adds unacceptable latency to packet reception and transmission.

Polling designs and interrupt-driven designs differ in their placement of policy decisions. When the behavior of tasks cannot be predicted, we rely on the scheduler and the interrupt system to dynamically allocate CPU resources. When tasks can be expected to behave in a predictable manner, the tasks themselves are better able to make the scheduling decisions, and polling depends on voluntary cooperation among the tasks.

Since a purely interrupt-driven system leads to livelock, and a purely polling system adds unnecessary latency, we employ a hybrid design, in which the system polls only when triggered by an interrupt, and interrupts happen only while polling is suspended. During low loads, packet arrivals are unpredictable and we use interrupts to avoid latency. During high loads, we know that packets are arriving at or near the system’s saturation rate, so we use polling to ensure progress and fairness, and only re-enable interrupts when no more work is pending.

5.3. Avoiding preemption

As we showed in section 4.2, receive livelock occurs because interrupt processing preempts all other packet processing. We can solve this problem by making higher-level packet processing non-preemptable. We observe that this can be done following one of two general approaches: do (almost) everything at high IPL, or do (almost) nothing at high IPL.

Following the first approach, we can modify the 4.2BSD design (see section 4.1) by eliminating the software interrupt, polling interfaces for events, and processing received packets to completion at device IPL. Because higher-level processing occurs at device IPL, it cannot be preempted by another packet arrival, and so we guarantee that livelock does not occur within the kernel’s protocol stack. We still

need to use a rate-control mechanism to ensure progress by user-level applications.

In a system following the second approach, the interrupt handler runs only long enough to set a “service needed” flag, and to schedule the polling thread if it is not already running. The polling thread runs at zero IPL, checking the flags to decide which devices need service. Only when the polling thread is done does it re-enable the device interrupt. The polling thread can be interrupted at most once by each device, and so it progresses at full speed without interference.

Either approach eliminates the need to queue packets between the device driver and the higher-level protocol software, although if the protocol stack must block, the incoming packet must be queued at a later point. (For example, this would happen when the data is ready for delivery to a user process, or when an IP fragment is received and its companion fragments are not yet available.)

5.4. Summary of techniques

In summary, we avoid livelock by:

- Using interrupts only to initiate polling.
- Using round-robin polling to fairly allocate resources among event sources.
- Temporarily disabling input when feedback from a full queue, or a limit on CPU usage, indicates that other important tasks are pending.
- Dropping packets early, rather than late, to avoid wasted work. Once we decide to receive a packet, we try to process it to completion.

We maintain high performance by

- Re-enabling interrupts when no work is pending, to avoid polling overhead and to keep latency low.
- Letting the receiving interface buffer bursts, to avoid dropping packets.
- Eliminating the IP input queue, and associated overhead.

We observe, in passing, that inefficient code tends to exacerbate receive livelock, by lowering the MLFRR of the system and hence increasing the likelihood that livelock will occur. Aggressive optimization, “fast-path” designs, and removal of unnecessary steps all help to postpone arrival of livelock.

6. Livelock in BSD-based routers

In this section, we consider the specific example of an IP packet router built using Digital UNIX (formerly DEC OSF/1). We chose this application because routing performance is easily measured. Also, since firewalls typically use UNIX-based

routers, they must be livelock-proof in order to prevent denial-of-service attacks.

Our goals were to (1) obtain the highest possible maximum throughput; (2) maintain high throughput even when overloaded; (3) allocate sufficient CPU cycles to user-mode tasks; (4) minimize latency; and (5) avoid degrading performance in other applications.

6.1. Measurement methodology

Our test configuration consisted of a router-under-test connecting two otherwise unloaded Ethernets. A source host generated IP/UDP packets at a variety of rates, and sent them via the router to a destination address. (The destination host did not exist; we fooled the router by inserting a phantom entry into its ARP table.) We measured router performance by counting the number of packets successfully forwarded in a given period, yielding an average forwarding rate.

The router-under-test was a DECstation™ 3000/300 Alpha-based system running Digital UNIX V3.2, with a SPECint92 rating of 66.2. We chose the slowest available Alpha host, to make the livelock problem more evident. The source host was a DECstation 3000/400, with a SPECint92 rating of 74.7. We slightly modified its kernel to allow more efficient generation of output packets, so that we could stress the router-under-test as much as possible.

In all the trials reported on here, the packet generator sent 10000 UDP packets carrying 4 bytes of data. This system does not generate a precisely paced stream of packets; the packet rates reported are averaged over several seconds, and the short-term rates varied somewhat from the mean. We calculated the delivered packet rate by using the “netstat” program (on the router machine) to sample the output interface count (“Opkts”) before and after each trial. We checked, using a network analyzer on the stub Ethernet, that this count exactly reports the number of packets transmitted on the output interface.

6.2. Measurements of an unmodified kernel

We started by measuring the performance of the unmodified operating system, as shown in figure 6-1. Each mark represents one trial. The filled circles show kernel-based forwarding performance, and the open squares show performance using the *screened* program [7], used in some firewalls to screen out unwanted packets. This user-mode program does one system call per packet; the packet-forwarding path includes both kernel and user-mode code. In this case, *screened* was configured to accept all packets.

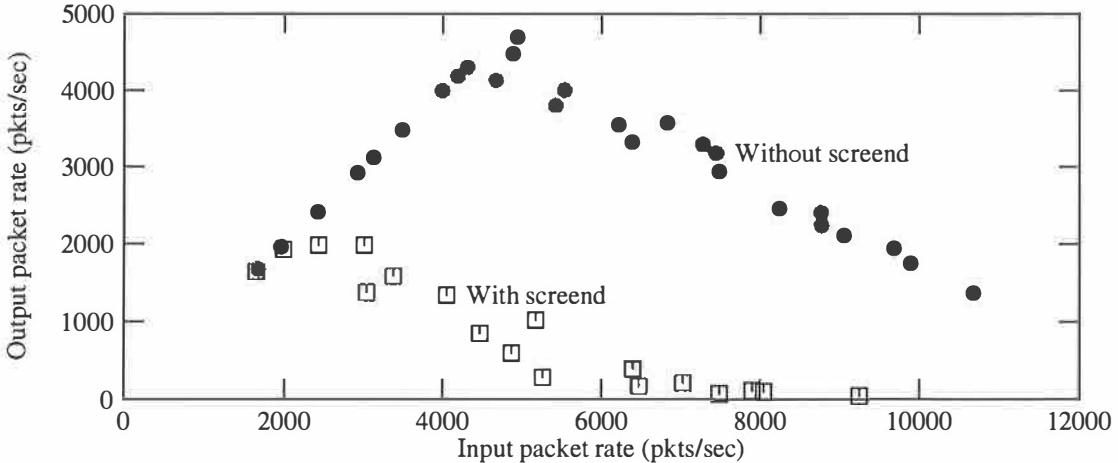


Figure 6-1: Forwarding performance of unmodified kernel

From these tests, it was clear that with *screend* running, the router suffered from poor overload behavior at rates above 2000 packets/sec., and complete livelock set in at about 6000 packets/sec. Even without *screend*, the router peaked at 4700 packets/sec., and would probably livelock somewhat below the maximum Ethernet packet rate of about 14,880 packets/second.

6.3. Why livelock occurs in the 4.2BSD model

4.2BSD follows the model described in section 4.1, and depicted in figure 6-2. The device driver runs at interrupt priority level (IPL) = SPLIMP, and the IP layer runs via a software interrupt at IPL = SPLNET, which is lower than SPLIMP. The queue between the driver and the IP code is named ‘*ipintrq*,’ and each output interface is buffered by a queue of its own. All queues have length limits; excess packets are dropped. Device drivers in this system implement interrupt batching, so at high input rates very few interrupts are actually taken.

Digital UNIX follows a similar model, with the IP layer running as a separately scheduled thread at IPL = 0, instead of as a software interrupt handler.

It is now quite obvious why the system suffers from receive livelock. Once the input rate exceeds the rate at which the device driver can pull new packets out of the interface and add them to the IP input queue, the IP code never runs. Thus, it never removes packets from its queue (*ipintrq*), which fills up, and all subsequent received packets are dropped.

The system’s CPU resources are saturated because it discards each packet after a lot of CPU time has been invested in it at elevated IPL. This is foolish; once a packet has made its way through the device driver, it represents an investment and should be processed to completion if at all possible. In a router, this means that the packet should be trans-

mitted on the output interface. When the system is overloaded, it should discard packets as early as possible (i.e., in the receiving interface), so that discarded packets do not waste any resources.

6.4. Fixing the livelock problem

We solved the livelock problem by doing as much work as possible in a kernel thread, rather than in the interrupt handler, and by eliminating the IP input queue and its associated queue manipulations and software interrupt (or thread dispatch)¹. Once we decide to take a packet from the receiving interface, we try not to discard it later on, since this would represent wasted effort.

We also try to carefully ‘‘schedule’’ the work done in this thread. It is probably not possible to use the system’s real scheduler to control the handling of each packet, so we instead had this thread use a polling technique to efficiently simulate round-robin scheduling of packet processing. The polling thread uses additional heuristics to help meet our performance goals.

In the new system, the interrupt handler for an interface driver does almost no work at all. Instead, it simply schedules the polling thread (if it has not already been scheduled), recording its need for packet processing, and then returns from the interrupt. It does not set the device’s interrupt-enable flag, so the system will not be distracted with additional interrupts until the polling thread has processed all of the pending packets.

At boot time, the modified interface drivers register themselves with the polling system, provid-

¹This is not such a radical idea; Van Jacobson had already used it as a way to improve end-system TCP performance [4].

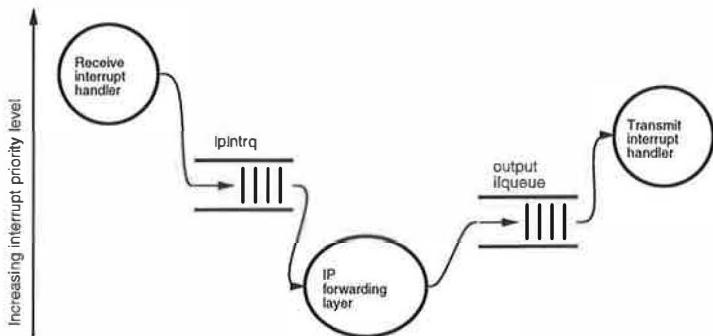


Figure 6-2: IP forwarding path in 4.2BSD

ing callback procedures for handling received and transmitted packets, and for enabling interrupts. When the polling thread is scheduled, it checks all of the registered devices to see if they have requested processing, and invokes the appropriate callback procedures to do what the interrupt handler would have done in the unmodified kernel.

The received-packet callback procedures call the IP input processing routine directly, rather than placing received packets on a queue for later processing; this means that any packet accepted from the interface is processed as far as possible (e.g., to the output interface queue for forwarding, or to a queue for delivery to a process). If the system falls behind, the interface's input buffer will soak up packets for a while, and any excess packets will be dropped by the interface before the system has wasted any resources on it.

The polling thread passes the callback procedures a quota on the number of packets they are allowed to handle. Once a callback has used up its quota, it must return to the polling thread. This allows the thread to round-robin between multiple interfaces, and between input and output handling on any given interface, to prevent a single input stream from monopolizing the CPU.

Once all the packets pending at an interface have been handled, the polling thread also invokes the driver's interrupt-enable callback so that a subsequent packet event will cause an interrupt.

6.5. Results and analysis

Figures 6-3 summarizes the results of our changes, when *screend* is not used. Several different kernel configurations are shown, using different mark symbols on the graph. The modified kernel (shown with square marks) slightly improves the MLFRR, and avoids livelock at higher input rates.

The modified kernel can be configured to act as if it were an unmodified system (shown with open circles), although this seems to perform slightly

worse than an actual unmodified system (filled circles). The reasons are not clear, but may involve slightly longer code paths, different compilers, or unfortunate changes in instruction cache conflicts.

6.6. Scheduling heuristics

Figure 6-3 shows that if the polling thread places no quota on the number of packets that a callback procedure can handle, when the input rate exceeds the MLFRR the total throughput drops almost to zero (shown with diamonds in the figure). This livelock occurs because although the packets are no longer discarded at the IP input queue, they are still piling up (and being discarded) at the queue for the output interface. This queue is unavoidable, since there is no guarantee that the output interface runs as fast as the input interface.

Why does the system fail to drain the output queue? If packets arrive too fast, the input-handling callback never finishes its job. This means that the polling thread never gets to call the output-handling callback for the transmitting interface, which prevents the release of transmitter buffer descriptors for use in further packet transmissions. This is similar to the transmit starvation condition identified in section 4.4.

The result is actually worse in the no-quota modified kernel, because in that system, packets are discarded for lack of space on the output queue, rather than on the IP input queue. The unmodified kernel does less work per discarded packet, and therefore occasionally discards them fast enough to catch up with a burst of input packets.

6.6.1. Feedback from full queues

How does the modified system perform when the *screend* program is used? Figure 6-4 compares the performance of the unmodified kernel (filled circles) and several modified kernels.

With the kernel modified as described so far (squares), the system performs about as badly as the

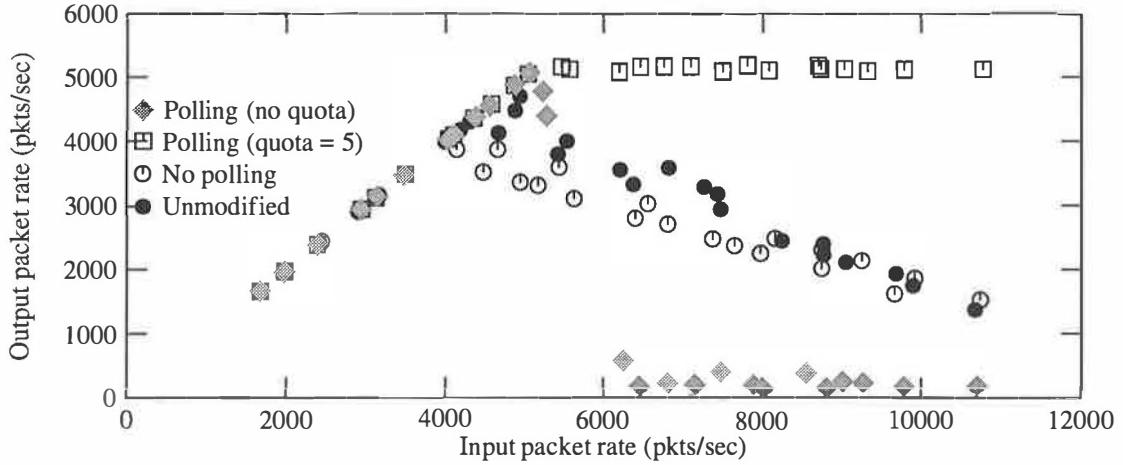


Figure 6-3: Forwarding performance of modified kernel, without using *screend*

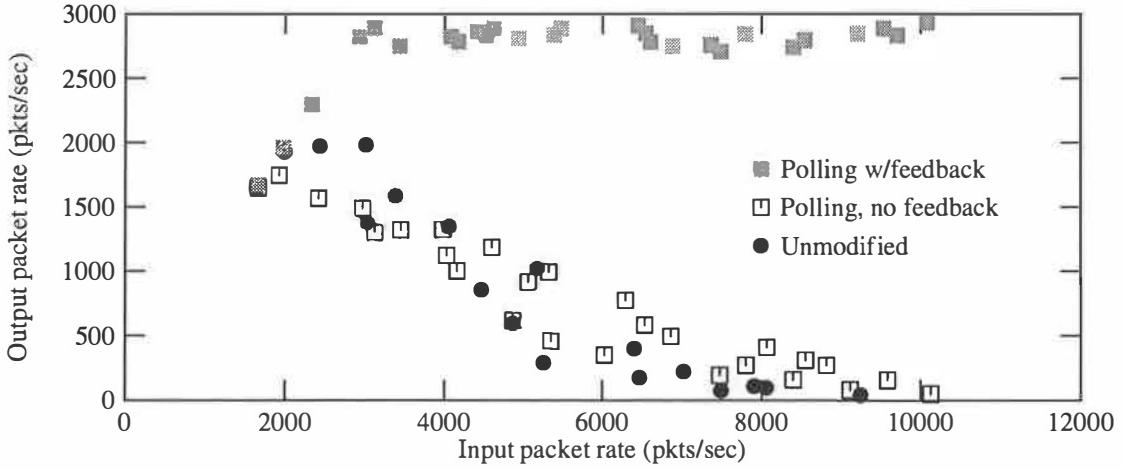


Figure 6-4: Forwarding performance of modified kernel, with *screend*

unmodified kernel. The problem is that, because *screend* runs in user mode, the kernel must queue packets for delivery to *screend*. When the system is overloaded, this queue fills up and packets are dropped. *screend* never gets a chance to run to drain this queue, because the system devotes its cycles to handling input packets.

To resolve this problem, we detect when the screening queue becomes full and inhibit further input processing (and input interrupts) until more queue space is available. The result is shown with the gray square marks in figure 6-4: no livelock, and much improved peak throughput. Feedback from the queue state means that the system properly allocates CPU resources to move packets all the way through the system, instead of dropping them at an intermediate point.

In these experiments, the polling quota was 10 packets, the screening queue was limited to 32 packets, and we inhibited input processing when the queue was 75% full. Input processing is re-enabled when the screening queue becomes 25% full. We chose these high and low water marks arbitrarily, and

some tuning might help. We also set a timeout (arbitrarily chosen as one clock tick, or about 1 msec) after which input is re-enabled, in case the *screend* program is hung, so that packets for other consumers are not dropped indefinitely.

The same queue-state feedback technique could be applied to other queues in the system, such as interface output queues, packet filter queues (for use in network monitoring) [9, 8], etc. The feedback policies for these queues would be more complex, since it might be difficult to determine if input processing load was actually preventing progress at these queues. Since the *screend* program is typically run as the only application on a system, however, a full screening queue is an unequivocal signal that too many packets are arriving.

6.6.2. Choice of packet-count quota

To avoid livelock in the non-*screend* configuration, we had to set a quota on the number of packets processed per callback, so we investigated how system throughput changes as the quota is varied. Figure 6-5 shows the results; smaller quotas work

better. As the quota increases, livelock becomes more of a problem.

When *screened* is used, however, the queue-state feedback mechanism prevents livelock, and small quotas slightly reduce maximum throughput (by about 5%). We believe that by processing more packets per callback, the system amortizes the cost of polling more effectively, but increasing the quota could also increase worst-case per-packet latency. Once the quota is large enough to fill the screening queue with a burst of packets, the feedback mechanism probably hides any potential for improvement.

Figure 6-6 shows the results when the *screened* process is in use.

In summary, tests both with and without *screened* suggest that a quota of between 10 and 20 packets yields stable and near-optimum behavior, for the hardware configuration tested. For other CPUs and network interfaces, the proper value may differ, so this parameter should be tunable.

7. Guaranteeing progress for user-level processes

The polling and queue-state feedback mechanisms described in section 6.4 can ensure that all necessary phases of packet processing make progress, even during input overload. They are indifferent to the needs of other activities, however, so user-level processes could still be starved for CPU cycles. This makes the system's user interface unresponsive and interferes with housekeeping tasks (such as routing table maintenance).

We verified this effect by running a compute-bound process on our modified router, and then flooding the router with minimum-sized packets to be forwarded. The router forwarded the packets at the full rate (i.e., as if no user-mode process were consuming resources), but the user process made no measurable progress.

Since the root problem is that the packet-input handling subsystem takes too much of the CPU, we should be able to ameliorate that by simply measuring the amount of CPU time spent handling received packets, and disabling input handling if this exceeds a threshold.

The Alpha architecture, on which we did these experiments, includes a high-resolution low-overhead counter register. This register counts every instruction cycle (in current implementations) and can be read in one instruction, without any data cache misses. Other modern RISC architectures support similar counters; Intel's Pentium is known to have one as an unsupported feature.

We measure the CPU usage over a period defined as several clock ticks (10 msec, in our current implementation, chosen arbitrarily to match the scheduler's quantum). Once each period, a timer function clears a running total of CPU cycles used in the packet-processing code.

Each time our modified kernel begins its polling loop, it reads the cycle counter, and reads it again at the end of the loop, to measure the number of cycles spent handling input and output packets during the loop. (The quota mechanism ensures that this interval is relatively short.) This number is then added to the running total, and if this total is above a threshold, input handling is immediately inhibited. At the end of the current period, a timer re-enables input handling. Execution of the system's idle thread also re-enables input interrupts and clears the running total.

By adjusting the threshold to be a fraction of the total number of cycles in a period, one can control fairly precisely the amount of CPU time spent processing packets. We have not yet implemented a programming interface for this control; for our tests, we simply patched a kernel global variable representing the percentage allocated to network processing, and the kernel automatically translates this to a number of cycles.

Figure 7-1 shows how much CPU time is available to a compute-bound user process, for several settings of the cycle threshold and various input rates. The curves show fairly stable behavior as the input rate increases, but the user process does not get as much CPU time as the threshold setting would imply.

Part of the discrepancy comes from system overhead; even with no input load, the user process gets about 94% of the CPU cycles. Also, the cycle-limit mechanism inhibits packet input processing but not output processing. At higher input rates, before input is inhibited, the output queue fills enough to soak up additional CPU cycles.

Measurement error could cause some additional discrepancy. The cycle threshold is checked only after handling a burst of input packets (for these experiments, the callback quota was 5 packets). With the system forwarding about 5000 packets/second, handling such a burst takes about 1 msec, or about 10% of the threshold-checking period.

The initial dips in the curves for the 50% and 75% thresholds probably reflect the cost of handling the actual interrupts; these cycles are not counted against the threshold, and at input rates below saturation, each incoming packet may be handled fast enough that no interrupt batching occurs.

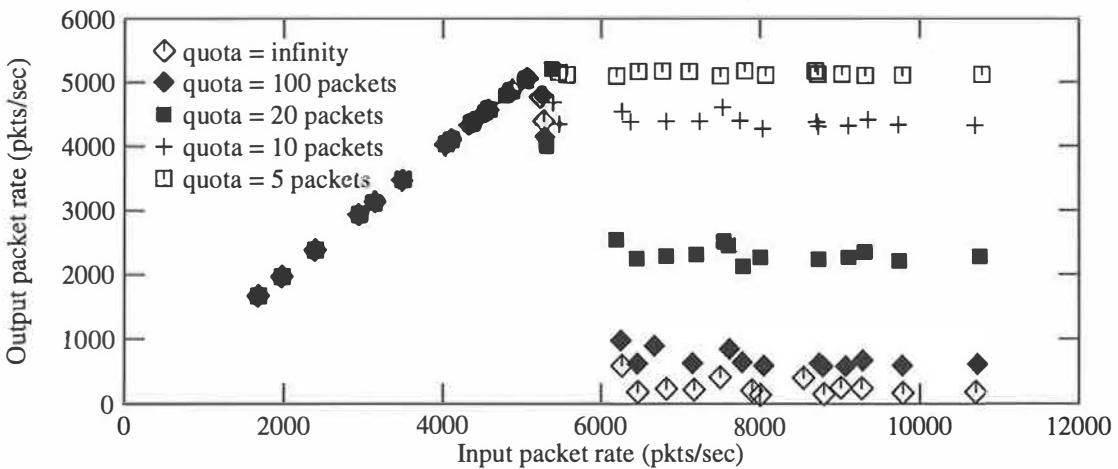


Figure 6-5: Effect of packet-count quota on performance, no *screend*

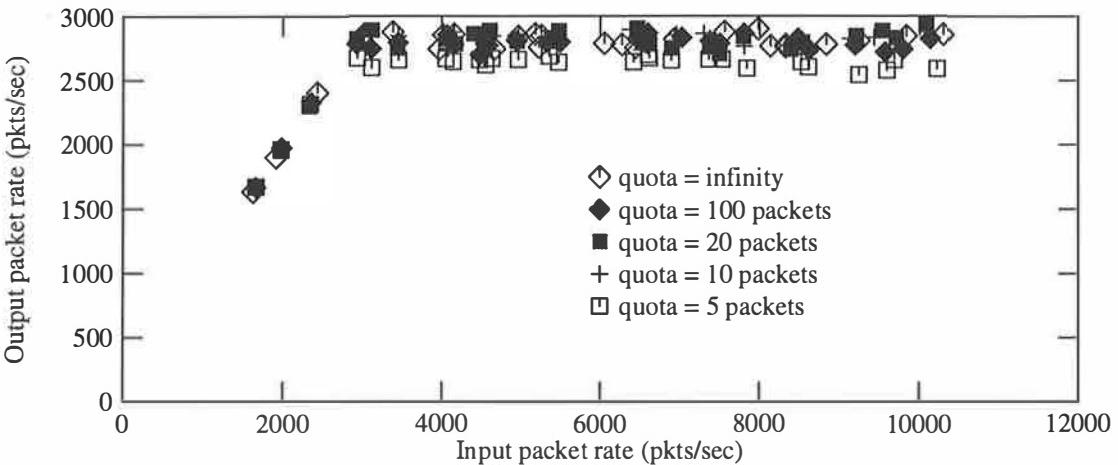


Figure 6-6: Effect of packet-count quota on performance, with *screend*

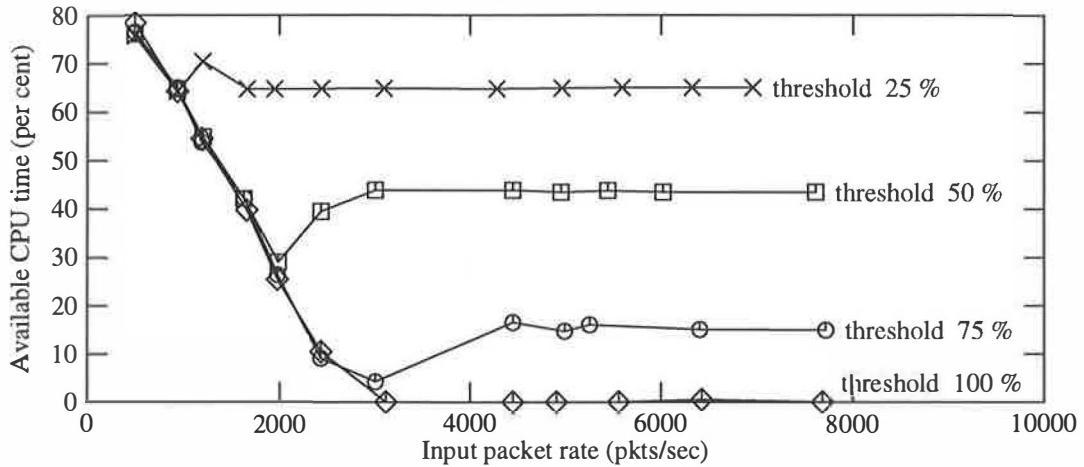


Figure 7-1: User-mode CPU time available using cycle-limit mechanism

With a cycle-limit imposed on packet processing, the system is subjectively far more responsive, even during heavy input overload. This improvement, however, is mostly apparent for local users; any network-based interaction, such as Telnet, still suffers because many packets are being dropped.

7.1. Performance of end-system transport protocols

The changes we made to the kernel potentially affect the performance of end-system transport protocols, such as TCP and the UDP/RPC/XDR/NFS stack. Since we have not yet applied our modifications to a high-speed network interface driver, such

as one for FDDI, we cannot yet measure this effect. (The test system can easily saturate an Ethernet, so measuring TCP throughput over Ethernet shows no effect.)

The technique of processing a received packet directly from the device driver to the TCP layer, without placing the packet on an IP-level queue, was used by Van Jacobson specifically to improve TCP performance [4]. It should reduce the cost of receiving a packet, by avoiding the queue operations and any associated locking; it also should improve the latency of kernel-to-kernel interactions (such as TCP acknowledgements and NFS RPCs).

The technique of polling the interfaces should not reduce end-system performance, because it is done primarily during input overload. (Some implementations use polling to avoid transmit interrupts altogether [6].) During overload, the unmodified system would not make any progress on applications or transport protocols; the use of polling, queue-state feedback, and CPU cycle limits should give the modified system a chance to make at least some progress.

8. Related work

Polling mechanisms have been used before in UNIX-based systems, both in network code and in other contexts. Whereas we have used polling to provide fairness and guaranteed progress, the previous applications of polling were intended to reduce the overhead associated with interrupt service. This does reduce the chances of system overload (for a given input rate), but does not prevent livelock.

Traw and Smith [14, 16] describe the use of “clocked interrupts,” periodic polling to learn of arriving packets without the overhead of per-packet interrupts. They point out that it is hard to choose the proper polling frequency: too high, and the system spends all its time polling; too low, and the receive latency soars. Their analysis [14] seems to ignore the use of interrupt batching to reduce the interrupt-service overhead; however, they do allude to the possibility of using a scheme in which an interrupt prompts polling for other events.

The 4.3BSD operating system [5] apparently used a periodic polling technique to process received characters from an eight-port terminal interface, if the recent input rate increased above a certain threshold. The intent seems to have been to avoid losing input characters (the device had little buffering available) but one could view this as a sort of livelock-avoidance strategy. Several router implementations use polling as their primary way to schedule packet processing.

When a congested router must drop a packet, its choice of which packet to drop can have significant effects. Our modifications do not affect *which* packets are dropped; we only change *when* they are dropped. The policy was and remains “drop-tail”; other policies might provide better results [3].

Some of our initial work on improved interface driver algorithms is described in [1].

9. Summary and conclusions

Systems that behave poorly under receive overload fail to provide consistent performance and good interactive behavior. Livelock is never the best response to overload. In this paper, we have shown how to understand system overload behavior and how to improve it, by carefully scheduling when packet processing is done.

We have shown, using measurements of a UNIX system, that traditional interrupt-driven systems perform badly under overload, resulting in receive livelock and starvation of transmits. Because such systems progressively reduce the priority of processing a packet as it goes further into the system, when overloaded they exhibit excessive packet loss and wasted work. Such pathologies may be caused not only by long-term receive overload, but also by transient overload from short-term bursty arrivals.

We described a set of scheduling improvements that help solve the problem of poor overload behavior. These include:

- Limiting interrupt arrival rates, to shed overload
- Polling to provide fairness
- Processing received packets to completion
- Explicitly regulating CPU usage for packet processing

Our experiments showed that these scheduling mechanisms provide good overload behavior and eliminate receive livelock. They should help both special-purpose and general-purpose systems.

Acknowledgements

We had help both in making measurements and in understanding system performance from many people, including Bill Hawe, Tony Lauck, John Poulin, Uttam Shikarpur, and John Dustin. Venkata Padmanabhan, David Cherkus, and Jeffry Yaplee helped during manuscript preparation.

Most of K. K. Ramakrishnan’s work on this paper was done while he was an employee of Digital Equipment Corporation.

References

- [1] Chran-Ham Chang, R. Flower, J. Forecast, H. Gray, W. R. Hawe, A. P Nadkarni, K. K. Ramakrishnan, U. N. Shikarpur, and K. M. Wilde. High-performance TCP/IP and UDP/IP Networking in DEC OSF/1 for Alpha AXP. *Digital Technical Journal* 5(1):44-61, Winter, 1993.
- [2] Domenico Ferrari, Joseph Pasquale, and George C. Polyzos. *Network Issues for Sequoia 2000*. Sequoia 2000 Technical Report 91/6, University of California, Berkeley, December, 1991.
- [3] Sally Floyd and Van Jacobson. Random Early Detection gateways for Congestion Avoidance. *Trans. Networking* 1(4):397-413, August, 1993.
- [4] Van Jacobson. Efficient Protocol Implementation. Notes from SIGCOMM '90 Tutorial on "Protocols for High-Speed Networks". 1990.
- [5] Samuel J. Leffler, Marshall Kirk McCusick, Michael J. Karels, and John S. Quarterman. *The Design and Implementation of the 4.3BSD UNIX Operating System*. Addison-Wesley, Reading, MA, 1989.
- [6] Rick Macklem. Lessons Learned Tuning The 4.3BSD Reno Implementation of the NFS Protocol. In *Proc. Winter 1991 USENIX Conference*, pages 53-64. Dallas, TX, January, 1991.
- [7] Jeffrey C. Mogul. Simple and Flexible Datagram Access Controls for Unix-based Gateways. In *Proc. Summer 1989 USENIX Conference*, pages 203-221. Baltimore, MD, June, 1989.
- [8] Jeffrey C. Mogul. Efficient Use Of Workstations for Passive Monitoring of Local Area Networks. In *Proc. SIGCOMM '90 Symposium on Communications Architectures and Protocols*, pages 253-263. ACM SIGCOMM, Philadelphia, PA, September, 1990.
- [9] Jeffrey C. Mogul, Richard F. Rashid, Michael J. Accetta. The Packet Filter: An Efficient Mechanism for User-Level Network Code. In *SOSP11*, pages 39-51. Austin, Texas, November, 1987.
- [10] Radia Perlman. Fault-Tolerant Broadcast of Routing Information. *Computer Networks* 7(6):395-405, December, 1983.
- [11] K. K. Ramakrishnan. Scheduling Issues for Interfacing to High Speed Networks. In *Proc. Globecom '92 IEEE Global Telecommunications Conf.*, pages 622-626. Orlando, FL, December, 1992.
- [12] K. K. Ramakrishnan. Performance Considerations in Designing Network Interfaces. *IEEE Journal on Selected Areas in Communications* 11(2):203-219, February, 1993.
- [13] Marcus J. Ranum and Frederick M. Avolio. A Toolkit and Methods for Internet Firewalls. In *Proc. Summer 1994 USENIX Conference*, pages 37-44. Boston, June, 1994.
- [14] Jonathan M. Smith and C. Brendan S. Traw. Giving Applications Access to Gb/s Networking. *IEEE Network* 7(4):44-52, July, 1993.
- [15] Robert J. Souza, P. G. Krishnakumar, Cuneyt M. Özveren, Robert J. Simcoe, Barry A. Spinney, Robert E. Thomas, and Robert J. Walsh. GIGAswitch: A High-Performance Packet Switching Platform. *Digital Technical Journal* 6(1):9-22, Winter, 1994.
- [16] C. Brendan S. Traw and Jonathan M. Smith. Hardware/Software Organization of a High-Performance ATM Host Interface. *IEEE Journal on Selected Areas in Communications* 11(2):240-253, February, 1993.

Jeffrey Mogul received an S.B. from the Massachusetts Institute of Technology in 1979, and his M.S. and Ph.D. degrees from Stanford University in 1980 and 1986. Since 1986, he has been a researcher at Digital's Western Research Laboratory, working on network and operating systems issues for high-performance computer systems. He is the author or co-author of several Internet Standards, an associate editor of *Internetworking: Research and Experience*, and was Program Chair for the Winter 1994 USENIX Conference.

Address for correspondence: Digital Equipment Corp. Western Research Lab, 250 University Ave., Palo Alto, CA, 94301 (mogul@wrl.dec.com)

K. K. Ramakrishnan is a Member of Technical Staff at AT&T Bell Laboratories. He holds a B.S. from Bangalore University in India in 1976, an M.S. from the Indian Institute of Science in 1978, and a Ph.D. from the University of Maryland in 1983. Until 1994, he was a Consulting Engineer at Digital. Ramakrishnan's research interests are in performance analysis and design of algorithms for computer networks and distributed systems. He is a technical editor for *IEEE Network Magazine* and is a member of the Internet Research Task Force's End-End Research Group.

Address for correspondence: AT&T Bell Laboratories, 600 Mountain Ave., Murray Hill, NJ, 07974 (kkrama@research.att.com)

UNIX is a registered trademark in the United States and other countries, licensed exclusively through X/Open Company, Ltd.

Windows NT is a trademark of Microsoft, Inc. GIGAswitch, VMS, and DECstation are trademarks of Digital Equipment Corporation.

Implementation of IPv6 in 4.4 BSD

Randall J. Atkinson, Daniel L. McDonald, Bao G. Phan,
Craig W. Metz*, & Kenneth C. Chin

Information Technology Division, Naval Research Laboratory

Abstract

The widespread availability of the TCP/IP protocols in early versions of BSD UNIX fostered the currently widespread use of those protocols in commercial products. Rapid depletion of the IPv4 address space has caused the Internet Engineering Task Force to design version 6 of the Internet Protocol (IPv6). IPv6 has some similarities with IPv4, but it also has many differences, most notably in address size. This paper describes our experience creating a freely distributable implementation of IPv6 inside 4.4 BSD, with focus on the areas that have changed between the IPv4 and IPv6 implementations.

1 Introduction

During the past decade, the worldwide Internet has grown at exponential rates, not only in North America but also in Europe and Asia. [Lot92] This, combined with suboptimal address allocation practices, has led to increasing depletion of the IP version 4 (IPv4) address space. One direct result of the IPv4 address depletion was that the Internet Engineering Task Force (IETF), began working to create a revised version of the Internet Protocol (IP). This effort is called Next-Generation IP (IPng). The resulting protocol is IP version 6 (IPv6). When the IPng effort began, there were several contenders, but in July 1994 the SIPP proposal became the primary basis for IPv6.

The widespread availability of TCP/IPv4 in early versions of BSD UNIX was crucial to the success and deployment of the Internet technologies. In order to help make Next-Generation IP as widely available, the authors began working with the Simple Internet Protocol (SIP) Working Group of the IETF in 1992.[Dee93] As SIP evolved into SIPP [Hin94] and then into IPv6, the authors began prototyping, initially in BSD Net/2 and currently in 4.4 BSD.

* Although Craig W. Metz is with Kaman Sciences Corporation, he may be reached at NRL.

Our primary development systems were Sun SPARC workstations and i486 systems running 4.4 BSD¹.

Implementation issues, rather than the details of the IPv6 protocol, are the focus of this paper. A number of implementation issues arose with IPv6 and have been resolved. Obvious issues, such as supporting 128 bit addresses instead of 32 bit addresses, are discussed in addition to the less obvious issues of how to implement IPv6 security inside a BSD kernel. We assume that the reader is somewhat familiar with the IPv6 protocol [DH95] and the 4.4 BSD-Lite implementation of IPv4. Figure 1 shows a rough overview of 4.4 BSD-Lite's Internet implementation, along with some of the new modules for IPv6. To add a new version of IP, many of the surrounding modules had to be modified as well.

2 Changes in Basic IP Functions

2.1 Differences in packet format

Perhaps the most obvious difference between IPv6 and its predecessor is the packet format. Although some in the Internet community felt that 64 bit addresses were sufficiently large, others insisted that 128-bit addresses were needed so that plug-and-play address assignment similar to ISO ES-IS could be supported. Many of the IPv4 header fields that were unused in practice (Figure 2) were eliminated or moved to options, making the IPv6 base header (Figure 3) more streamlined. One significant addition to the header is the *Flow Identifier* which is an important hook for resource reservation techniques [ZBE⁺93] currently being developed within the IETF.

The sparse IPv6 header is optimized for minimal processing. An IPv6 router needs only to verify the version number, inspect the destination address,

¹The systems running 4.4 BSD (encumbered) have had the 4.4 BSD-Lite networking changes incorporated into them. Some call this a BSD Net/3 system.

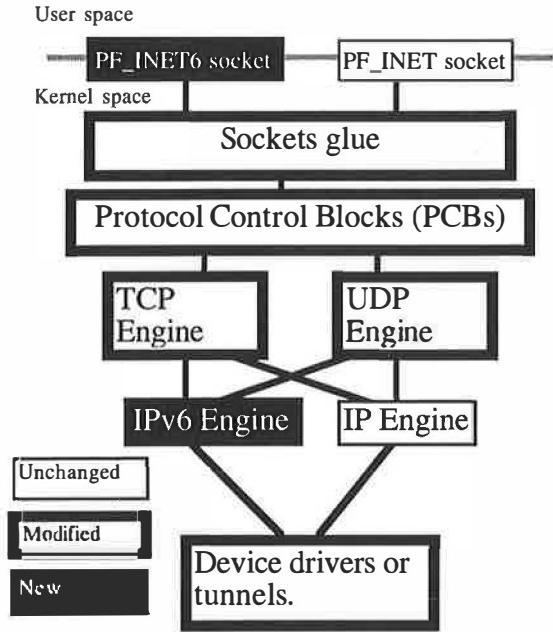


Figure 1: Simple Overview of 4.4 BSD-Lite Internet Modules

decrement the hop counter, and process hop-by-hop options if they are present. (The flow label can be used to optimize this process further.) An IPv4 router has to perform everything an IPv6 router does, as well as verify and recompute the header checksum, and fragment the datagram further if needed. An IPv6 destination host initially only has to check the validity of the version and destination address. If there are options, they are daisy-chained and indicated by the Next Header field. Otherwise, a higher-level protocol (e.g. TCP) is the next header processed. An IPv4 destination host has to verify not only the version and destination address, but the IP header checksum as well.

2.2 Protocol Processing

A number of the more recently developed IPv4 optional features are mandatory in IPv6. Other features, such as cryptographic security, are new with IPv6². These have caused a number of changes in IP protocol processing.

IPv6 daisy-chains optional headers after the base header. Our implementation pre-parses an IP packet into its constituent headers and upper-layer proto-

²The cryptographic security recently standardised for IPv4 and IPv6 was originally designed for use with IPv6 and later adapted for use with IPv4.

col data as part of the initial IPv6 input processing. Although this does degrade performance, it has simplified the processing of optional IPv6 headers. We plan to create a fast path around the preparsing code for packets containing no optional headers.

The Path MTU Discovery [MD90] technique for avoiding IP fragmentation in routers is mandatory for IPv6. IPv6 does not have any intermediate fragmentation and instead relies on Path MTU Discovery and end-to-end fragmentation. Our implementation stores Path MTU information in host routes. Host routes are automatically created for IP communications originating on the local machine. Storing this information in the routing table makes this data available to TCP, UDP, and ICMP. IPv6 requires a minimum MTU of 576 bytes, which is much larger than the 68 byte minimum MTU of IPv4. However, even this larger size might be too small if certain IPv6 options, such as the Hop-by-Hop Options Header (which can be up to 2048 octets), are used. In such cases, end-to-end fragmentation will be required.

3 Security Processing

Two cryptographic security mechanisms have been defined for IPv6 [Atk95c]. One, known as the Authentication Header (AH), provides authentication without confidentiality [Atk95a]. The second, known as the Encapsulating Security Payload (ESP), provides confidentiality through encryption of packet contents. [Atk95b] ESP has two modes. The first mode, known as *Transport-mode*, encrypts only the upper-layer header and data (such as TCP, UDP, or ICMP) and leaves the IP header in the clear. The second mode, known as *Tunnel-mode*, encrypts an entire IP datagram, prepending an additional clear-text IP header outside the encrypted IP datagram so that the packet can be routed. The implementation of these mechanisms broke new ground within the BSD kernel. In addition to implementing the Authentication Header and both modes of ESP, we also implemented the kernel support required to manage network security associations, including the cryptographic keys.

The IPv6 security mechanisms can use any appropriate encryption or authentication algorithm. The mandatory algorithms for a compliant implementation are keyed MD5[MKS95b] for authentication, and DES-CBC[MKS95a] for encryption. Both algorithms are in this implementation. To implement a new ESP or AH algorithm, the kernel must be recompiled with support for the new algorithms in place. Other algorithms, such as triple-DES, are be-

Version	Hdr Len	Type of Service	Packet Length
Fragment Identification		Flags and Fragment Offset	
Time-To-Live	Protocol	Header Checksum	
Source Address			Destination Address

Figure 2: IPv4 Packet Format

Version	Priority	Flow Label		
Payload Length		Next Header	Hop Limit	
Source Address				
Destination Address				

Figure 3: IPv6 Packet Format

ing implemented by others. Later in this paper, we discuss why it is straightforward to add support for additional cryptographic algorithms.

Both ESP Transport-mode encryption and Authentication Header output processing are normally performed immediately before any fragmentation on outgoing packets and after reassembly on the input side. They are done this way because, except for fragmentation, they need to operate on the packet as it will appear on the wire. For example, the source address for the packet from a multi-homed system must be known before encryption or authentication can take place.

3.1 Security Associations

A fundamental concept behind IP security is the *Security Association*. A Security Association contains all of the configuration data for a particular secure session between two or more systems communicating via IP. For example, the security services in use (AH or ESP), the cryptographic algorithm(s) in use, the cryptographic key(s) in use, the key lifetimes, the Security Parameters Index (SPI), and the sensitivity level (e.g. Unclassified, Secret) of the session are all components of a Security Association. In order to support multicast as well as unicast, all Security Associations are one-way from source to destination. So a typical telnet session would need two Security Associations, one in each direction.

Security associations are stored in a table inside the kernel. A module called the *Key Engine* controls access to the table. The Key Engine allows kernel services, such as the IPv6 module, to obtain secu-

rity associations for inbound and outbound packets. The Key Engine also communicates with user-level key management programs so that key management may be implemented properly. The relationship between the key engine and user-level key management programs is similar to the relationship between the routing socket[Sk191] and programs such as gated(8).

3.2 Security Processing Structure

The authentication processing function is split into three major parts. The first, a keyed message digest function, is selected on a per-association basis through an *algorithm switch* that calls the appropriate computation function. The second, the header processing routines, finds the appropriate security association and policy actions for the packet and either builds or parses the actual option header for authentication. The third part is the meat of the authentication function. This routine walks the packet, header by header, zeroing header fields that vary unpredictably end-to-end, and passing other header fields and the packet data into the keyed message digest function. The resulting message digest data can be either inserted into the outgoing header or, in the case of an incoming packet, checked with the one in the header. The keyed message digest functions are treated in the AH calculation function as stream operations; any necessary blocking and padding must be handled by the implementation of the keyed message digest functions.

The encryption processing function is split into similar parts. The first, an encryption/decryption function, and the second, a transform header con-

struction and parsing function, are selected on a per-association basis through an algorithm switch. Because almost all of the header format can vary depending on which cryptographic transform is being used, it is necessary that both the cryptographic functions and the header processing functions be switchable. There is a generic reblocking function that runs a specified encryption or decryption function over the data while arranging it into properly sized blocks. Block-oriented encryption and decryption functions require the encrypted data to be an integral number of cryptographic blocks.

3.3 Output Security Processing

Immediately before IP fragmentation is performed, `ipv6_output()` calls an IP security output policy function, `ipsec_output_policy()`, to determine whether this packet needs security. This function examines the system security level configured by the administrator and the socket security level requested by the process on the socket. The function is able to examine the socket security level because each outgoing packet data chain now contains a back pointer to the socket that sent the packet. The security output policy function then examines the system-wide security policy and the socket-requested security policy and applies the more paranoid of these policies to the outgoing packet.

The `ipsec_output_policy()` function is also responsible for making the `getassocbysocket()` call into the *Key Engine* to obtain *Security Association* data for the outgoing packet. If the *Key Engine* has the appropriate *Security Associations*, it provides access to them. If no appropriate *Security Association* exists and a key management daemon is running, then the *Key Engine* sends a *Request* message to that daemon and informs the output policy function that the *Security Association* has been delayed. If no appropriate *Security Association* exists and no key management daemon is running, then the *Key Engine* returns an error to `ipsec_output_policy()`. If this error occurs, it will eventually be presented to the user as the newly defined IP Security processing error, **EIPSEC**.

If IP security is needed and all appropriate security information is available for the outgoing packet, then the output security policy function will return both an indication of which services are needed and pointers to the appropriate *Security Associations*. The IP Output function then makes the appropriate calls to apply outgoing security services and then sends the packet out. If any errors occur during security output processing, the packet will be dropped

and the user will be given the **EIPSEC** error mentioned above. In the future, we might enhance the `getassocbysocket()` call to provide the user identification or uid associated with the network socket so that the *Key Engine* can provide finer granularity of keying. The current implementation does support both shared (i.e. host-oriented) keys and also unique (i.e. socket-oriented) keys.

3.4 Input Security Processing

For incoming packets, the task is significantly easier. When an Authentication Header or Encapsulating Security Payload header is encountered, it is processed by calling the appropriate IP security input function (either `ipsec_ah_input()` or `ipsec_esp_input()`). That function reads the Security Parameters Index (SPI) contained in the clear-text portion of the received packet and makes a `getassocbyspi()` call into the *Key Engine* to obtain the correct *Security Association* for the received packet. If this call succeeds, the security input processing is performed and the appropriate security-related flag is set. The packet data chain has two new flags, both initially cleared on input, called **M_AUTHENTIC** and **M_DECRYPTED**. These flags indicate that the packet passed authentication processing and encryption processing, respectively. If any security input processing fails, the packet is dropped and appropriate kernel statistics counters are incremented. A modified `netstat(8)` is supplied that can display these statistics for the system administrator. If more than one form of security has been applied, then the packet will go through more than one security input processing function.

The input security processing code also performs special checks comparing the outer IP source address and the (previously encrypted) inner IP source address for the case when an IP datagram is tunneled inside another IP datagram and either the Authentication Header or the Encapsulating Security Payload is present. These checks are intended to prevent an adversary system from encapsulating a forged packet inside an authenticated or encrypted legitimate packet and tricking the receiving system into believing the forged packet was authentic. If these source address checks fail, then the **M_AUTHENTIC** or **M_DECRYPTED** flags on the received packet data chain are cleared.

After security input processing is completed, the normal input processing resumes. Once the packet reaches the transport layer, the transport layer's input function, for example `tcp_input()`, calls `ipsec_input_policy()` to perform an input secu-

rity policy check. The incoming packet is dropped if it does not meet the requirements for authentication or encryption that exist for its destination socket. Because `ipsec_input_policy()` checks not only the socket security requirements but also the system-wide security requirements, the system administrator can mandate a minimum security level for all normal network connections.

3.5 Policy Separation

The separation of the policy engine from the mechanisms allows per-socket security selections and administrative security selections to be combined in sophisticated ways. For instance, an administrator could require that packets coming in on a certain range of privileged ports must come from a privileged port and must be authentic in order to protect the administrator's system from potential abuses. The current policy engine only implements simple system-wide decisions (e.g., drop all non-authentic packets, always use authentication if we have a security association that will facilitate it) in conjunction with application requested socket security. Enhancements to the security policy engine are planned for the future.

3.6 Algorithm-independence

Care was taken to provide multiple levels of indirection to take advantage of the algorithm-independent nature of the Authentication Header and Encapsulating Security Payload (ESP) specifications. Both implementations use an algorithm switch, which is indexed by a value in the security association, to support multiple algorithms concurrently and allow easy addition of new message digest and encryption functions. This switch is more complex for ESP, because almost all of the ESP header format can change as a function of the transform in use. For this case, the switch allows implementors to specify the header processing code and the encryption code separately for greater flexibility. For instance, someone wanting to substitute the IDEA algorithm [LM91] for the default DES-CBC algorithm but still use the same basic header format could create a new algorithm switch entry that uses the same header processing functions as DES-CBC [MKS95a] but calls the IDEA encryption functions instead. Different algorithms will have different performance impacts. Supporting multiple algorithms in the kernel does not exact a significant performance penalty.

4 Changes to ICMP and IGMP

The Internet Control Message Protocol (ICMP) is perhaps not as widely known as TCP or UDP, but it performs a critical function in keeping the network operating smoothly. The Internet Group Membership Protocol (IGMP) is integral to IP multicasting. ICMP for IPv6 is sufficiently different that it is now sometimes referred to as ICMPv6 [Pos81][DC95].

Despite having similar header syntax, ICMPv6 differs from ICMP for IPv4 in four major ways. First, ICMPv6, like TCP and UDP, requires a pseudo-header to be included in its checksum calculation. Second, the difference between informational messages (e.g. *Echo*) and error messages (e.g. *Port Unreachable*) is now indicated by the high bit in the ICMPv6 message type. Third, ICMPv6 absorbs the functions of the formerly separate IGMP [Dee89], ARP [Plu82][FMMT84], Proxy ARP, and ICMP Router Discovery [Dee91] protocols. Finally, ICMPv6 also adds support for stateless address auto-configuration. Because ICMP is above the IP layer, all of these functions can now be authenticated and or encrypted using the IP security mechanisms, as long as appropriate security associations exist. Sites that wish to bootstrap securely can now do so.

4.1 Traditional ICMP and IGMP

ICMPv6 retains the functions traditionally performed by ICMP and IGMP. The *Echo* and *Echo-Reply* messages, utilized by ping(8), are still part of ICMPv6. Unreachability of varying forms is indicated by the ICMPv6 *Unreachable* message type. Extensions have been added to indicate unreachable on-link neighbors, as well as errors with strict source routing. A *Message Too Big* message indicates when an IPv6 datagram is too large for a link on its path. Path MTU discovery [MD90], a requirement for IPv6, is implemented using these messages. *Parameter Problem* messages indicate invalid IPv6 option fields, as they do in IPv4's ICMP. *Time Exceeded* messages indicate either a hop limit that has decremented to zero, or that an IPv6 reassembly has timed out.³

ICMPv6 has three additional informational messages: *Group Report*, *Group Query*, and *Group Terminate*. The first two behave just like the IGMP *Report* and *Query* messages. The *Group Terminate*

³This implementation cannot send Time Exceeded messages for IPv6 reassembly timeouts; the "offending packet" needed for the ICMPv6 message is no longer available for transmission because reassembly is occurring.

message is an optimization so that routers can be informed more quickly about hosts leaving multicast groups.

4.2 Address Auto-Configuration and Router Discovery

The Internet community mandated that IPv6 support simple address auto-configuration for hosts. IPv6 has two solutions to this problem. The first approach is to use an optional configuration protocol, such as DHCPv6. This solution is beyond the scope of this paper. The second approach, known as stateless address autoconfiguration, is required, and is implemented in ICMPv6 [TN95].

4.2.1 Link-local Addresses

When an interface is configured for IPv6, it must have a *link-local* address. A link-local address is formed by placing a link-local prefix **fe80::** in front of a token, usually the interface's MAC address. In our implementation, this is done by the ifconfig(8) application placing this address on an interface before any other addresses are placed on the same interface. Implementations must be able to detect whether their link-local address has been duplicated on the same link (e.g. Ethernet).[NNS95] Our planned approach to this collision detection is discussed in the Neighbor Discovery section. Once the link-local address is verified as being unique on a link, the first phase of stateless address auto-configuration is completed. The IPv6 node can then send out ICMPv6 *Router Solicit* messages to locate a router, and begin the second phase of address auto-configuration.

4.2.2 Router Discovery

IPv6 routers send out periodic *Router Advertisement* messages to the all-nodes multicast address. Also, IPv6 routers send out *Router Advertisement* messages in response to *Router Solicit* messages. Besides performing the traditional jobs of IPv4 router advertisements, IPv6 router advertisements also advertise parameters relating to Neighbor Discovery: suggested MTUs on variable-MTU links, suggested maximum hop limits, and on-link prefixes.

It is the advertisement of on-link prefixes which completes stateless address auto-configuration. If the *Router Advertisement* message indicates that stateless configuration is to be performed, the message will also contain the globally routable address prefix used on the link. The node then takes the token from its link-local address, and prepends the

advertised prefix to form an automatically configured globally routable address. The internal code to handle such advertisements also handles the manual address configuration requests from programs such as ifconfig(8).

Unlike IPv4, IPv6 addresses can have lifetimes. In concert with stateless address auto-configuration, lifetimes provide a way for relatively rapid IPv6 address renumbering to occur. Provider-oriented addressing is one of the address schemes that will be used with IPv6.[RLH⁺95] With provider-oriented addressing, the ability to rapidly renumber many systems at a site is essential if that site should ever want to change network service providers. Hence, IPv6 interface addresses in the kernel now contain lifetime fields.

4.3 Neighbor Discovery

IPv6 does not use ARP.⁴ Instead, IPv6 uses multicasting and ICMPv6 to discover the addresses of on-link neighbors.[NNS95] Our implementation uses host routes for on-link neighbors and keeps link-layer information inside the route, much as 4.4BSD implements ARP entries. Like ARP, IPv6 neighbor discovery has the route's gateway address point to a data-link socket address, for example an Ethernet MAC address.

IPv6 Neighbor Discovery is responsible for finding the link address information for the host route entries. If an IPv6 destination is determined to be on link, either by matching an on-link prefix (represented as a cloning network route, as IPv4 does), or by determining that there is no other way to reach a destination, a neighbor solicit is sent out to a special multicast address. The special multicast routing prefix **ff02::1:** is prepended to the low 32 bits of the solicited neighbor. All nodes automatically join the *Solicited Nodes* multicast group appropriate for their own addresses. Broadcast does not exist in IPv6; multicast replaces all uses for broadcast.⁵ Once a *Neighbor Solicit* is heard, enough information is known to send a unicast *Neighbor Advertisement* to the solicitor, and now the soliciting node knows that the neighbor is reachable. While the solicited node has enough information to return the unicast neighbor advertisement, reachability the opposite way is not yet confirmed. Unicast solicit and advertisement messages confirm the reachability of the neighbor after initial reachability is established. Upper-level protocols (e.g. TCP) can also be used

⁴Hence, ARP-related broadcast storm problems will not be present with IPv6

⁵Hence, broadcast storms will not exist with IPv6.

to provide reachability confirmation.⁶

Users can use `netstat -r` to examine the state of currently reachable and recently reachable neighbor systems. This neighbor reachability information is kept as part of the routing table in the kernel, so reachability updates for one session to a neighbor will also refresh reachability for other sessions to the same neighbor. Neighbors that have become unreachable will linger in the routing table and will eventually be marked with the `RTF_REJECT` flag. This is similar to the way ARP is handled in 4.4-Lite BSD.

Neighbor discovery can be used to detect the uniqueness of a link-local address. After a link-local address is configured, the node sends a multicast neighbor solicit for its proposed link-local address. If no neighbor responds with a neighbor advertisement, then the link-local address is unique for the link. The alpha release does not currently implement collision detection, because of the difficulty in placing the functionality of the detection. If done in the kernel, a user process may be trapped in the `ioctl(2)` call for a long time while collision detection takes place. If done in user space, multiple calls will have to be made into the kernel.

5 Transport Layer Changes

Both the UDP and TCP protocols remain unchanged for IPv6. However, the BSD implementations required modification to provide concurrent support for IPv4 and IPv6. The main difficulties arose due to the different sizes of the IPv4 header and the IPv6 header. Because the TCP and UDP implementations are shared between IPv4 and IPv6, we designed a modified Protocol Control Block (PCB) structure that supports both versions of IP. Had the original BSD implementation of TCP, UDP, and IP not been so closely coupled, it would have been easier to add IPv6 support into the kernel.

5.1 Protocol Control Block

Since TCP and UDP do not change between IPv4 and IPv6, TCP and UDP use the modified Protocol Control Block structures (PCBs) in the same way. With IPv6's larger address space, the PCBs were modified to support both IPv4 and IPv6 addresses and to denote which addresses are actually in use. To support both protocols, new unions were devised. To make these changes invisible to existing code, appropriate `#defines` were added that silently derefer-

⁶We are still experimenting with the best way for TCP to update reachability without impairing performance.

enced the appropriate component of the union. Figure 4 shows an example of a new union and its corresponding new `#defines`.

```
union {
    struct route ru_route;
    struct route6 ru_route6;
} inp_ru;

#define inp_route inp_ru.route
#define inp_route6 inp_ru.route6
```

Figure 4: Route union used in new PCB structure

The IPv4-IPv6 transition specification [GN95] makes it easier to support both protocols in a single PCB by allocating a portion of the IPv6 address space for use as "IPv4-mapped" addresses, which cannot be used as addresses in IPv6 datagrams. Additionally, if a session is intending to send IPv6 datagrams, a bit in the session's PCB's flags will be set indicating this. If that bit is not set, then IPv4 is in use. The route, IP header template, and multicast options elements now use unions so that either IPv4 or IPv6 can be used with the PCB.

New PCB functions were written to support bind, connect, and notify functions on `PF_INET6` sockets. Because such a socket can be used to send and receive either IPv4 or IPv6 traffic, these functions needed to be separate from the equivalent IPv4 functions and also needed to handle both versions of IP. In the near future we intend to enhance these functions to fully support the IPv6 *Flow Identifier* field so that real-time and predictive services are provided to applications. The `in6_pcbsnotify()` function also calls the input security policy function to determine whether a particular error can be passed upwards to the application or whether that would cause a security violation and the error should not be delivered.

5.2 Changes in UDP

The UDP protocol remains unchanged for IPv6, but the BSD implementation needed to be modified to support both versions of IP. The majority of the changes to the UDP code resulted from the need to support the different address format. The changes are minimal and are isolated to the following functions `udp_input()`, `udp_output()`, `udp_ctlinput()`, and `udp_usrreq()`. Almost all changes occur in the input and output processing of UDP datagrams, handled by the functions `udp_input()` and `udp_output()`, respectively.

Incoming UDP datagrams, regardless of whether they are transported over IPv4 or IPv6, are processed by `udp_input()`. Where the code needs to access elements of the IP header, different code paths are executed for IPv4 and IPv6 datagrams. The function relies on a local variable, which it sets on entrance to the function, to determine which code path to follow. An example of a code path specific to IPv6 is the processing of an IPv4 packet destined for an IPv6 socket. The IPv6 BSD Sockets API specification allows an application to receive both IPv4 and IPv6 datagrams using an IPv6 socket.[GTB95] Code has been added to allow `udp_input()` to handle this special case.

The `udp_input()` function now calls the input security policy function before processing an incoming packet. This ensures compliance with both socket and system security requirements. If an incoming packet should not be delivered for security policy reasons, then it is silently dropped. This check does exact a performance penalty on each received packet, but we have not yet found a better way to handle input security policy checks.

The function `udp_output()` is called to create and send a UDP datagram. It determines whether to create an IPv4 or IPv6 datagram by looking at the protocol control block for the socket originating the datagram. If the socket's protocol family is `PF_INET6` and the socket's PCB indicates that the destination is a native IPv6 address, an IPv6 UDP datagram is composed and sent down to the IP layer via the `ipv6_output()` function. If the protocol family is `PF_INET`, `ip_output()` is called instead of `ipv6_output`. A significant change in `udp_output()` from its IPv4 version involves the calculation of the UDP checksum. In IPv4, calculation of the UDP checksum is optional and is controlled by the global variable `udpcksum`. Since IPv6 no longer has an IP layer checksum, the UDP checksum is not optional and must be calculated for all IPv6 UDP packets. This is necessary to provide integrity protection of the source and destination address that is not provided by IPv6, which lacks an IP header checksum.

The remaining changes in `udp_ctlinput()` and `udp_usrreq()` are minor changes to call IPv6 versions of certain IPv4 functions or to initialize IPv6 specific variables in the protocol control block. Overall, the modifications of UDP code to work with both IPv4 and IPv6 are straightforward.

5.3 Changes in TCP

The TCP protocol also remains unchanged for IPv6, but was modified to support both versions of IP.

One change was to add a new member, `pf`, to the TCP control block structure, `struct tcpcb`. This new member stores the *Protocol Family*, either `PF_INET` for IPv4 or `PF_INET6` for IPv6, in use for each TCP session. This is used in several parts of the TCP code to help select the correct IP-specific code branch.

The beginning of the `tcp_input()` function has a small amount of IP-related processing. This was broken into two code paths, one for IPv4 and one for IPv6 at the cost of an if check and a slight increase in code size.

The main difficulty with the 4.4 BSD-Lite TCP implementation was its reliance on a single pointer, `struct tciphdr *ti`, that pointed to a structure containing both the IPv4 overlay header (Figure 5) and also the TCP header of received segments. The `tcp_input()` and `tcp_reass()` functions used this combined structure for most of the data references relating to a given TCP segment. There were also other uses of this structure within the TCP implementation. Because of the differing IP header sizes, the TCP header starts at a different offset from the start of the structure, depending on which IP header is present. The solution to this problem was to create a new pointer `struct tcphdr *th` which is calculated separately for IPv4 and IPv6, but always points to the TCP header. The references to TCP header data that had previously used `*ti` now use `*th` instead.

However, use of the `*th` pointer did not solve all of the problems. The older `struct tciphdr` contains an element `ti->tlen` that pointed to the packet's length field. There is not room to store such a data item in the `struct tcip6v6hdr`, which uses a `struct ipv6ovly` (Figure 6), but fortunately there was an existing local variable `tlen` in `tcp_input()` that is used instead. Most of the references to IP data elements are made at the very beginning of the `tcp_input()` function and so were easily handled.

The `tcp_reass()` function was not amenable to supporting both versions of IP at the same time, so our implementation increases code size by adding a new `tcpv6_reass()` function that uses `struct tcip6v6hdr` in lieu of the `struct tciphdr` used by the original `tcp_reass()`.

The `tcp_input()` function now calls the input security policy function before processing an incoming TCP segment. This ensures compliance with both socket and system security requirements. If an incoming segment should not be processed for security policy reasons, then it is silently dropped. If the system security policy is to require authen-

	ih_next (pointer to next segment hdr)
	ih_prev (pointer to prev segment hdr)
ih_x1 (pad)	ih_pr (protocol)
	ih_src (source address)
	ih_dst (destination address)

Figure 5: Format of **struct ipovly** IPv4 Overlay

ih_next (pointer to next segment header)
ih_prev (pointer to prev segment header)
ih_src (source address)
ih_dst (destination address)

Figure 6: Format of **struct ipv6ovly** IPv6 Overlay

tication on all received packets, then attempts to open an unauthenticated TCP connection or unauthenticated ping will silently fail as if the destination system were not reachable at all. As with the UDP implementation, this check exacts a performance penalty.

One benefit of our changes has been to isolate the network-layer code more. This might make it easier to modify TCP further to support TCP over other network-layer protocols, for example Novell's IPX. We are concerned about the adverse performance impact of the IPv6 changes, so we are examining methods of improving the performance of our implementation. We have not found anything in the IPv6 specifications that inherently reduces TCP performance.

6 Changes to Applications

6.1 Network Socket Enhancements

Although the IETF does not standardise application programming interfaces, some members of the IPng Working Group did create an Informational RFC describing how IPv6 might be used in conjunction with BSD Sockets [GTB95]. Some changes in 4.4-Lite BSD were needed to comply with that specification. Fortunately, most of the changes involved adding protocol switch tables, and entries to those tables[LMKQ89]. Other sockets changes were implemented at lower levels, most notably the aforementioned PCB code. One can use a PF_INET6 socket to communicate using IPv4 or IPv6, which makes it easier to transition applications to the new version

```
#include <sys/socket.h>
#include <netinet6/in6.h>
...
struct sockaddr_in6 addr6;
int s;

...
s = socket(PF_INET6, SOCK_DGRAM, 0);
addr6.sin6_len = sizeof(addr6);
addr6.sin6_family = AF_INET6;
addr6.sin6_port = htons(7);
addr6.sin6_flowinfo = 0;
(void) ascii2addr(AF_INET6,
                  "FE80::800:dead:beef",
                  &addr6.sin6_addr);
sendto(s, "hello", 6, 0, &addr6,
       sizeof(addr6));
...

```

Figure 7: Code fragment illustrating use of UDP over IPv6

of IP.

More extensive changes were needed to permit applications to request security services from IPv6. Several new socket options were defined and implemented, including SO_SECURITY_ENCRYPTION_TRANSPORT, SO_SECURITY_ENCRYPTION_TUNNEL, and SO_SECURITY_AUTHENTICATION. These new socket options are used by an application to request that ESP in transport-mode, ESP in tunnel-mode, or the

Authentication Header be used with this network session. Each also has an associated *Security Level* parameter. There are currently 4 security levels implemented. Level 0 does not use security on outbound packets and does not require it on inbound packets. Level 1 uses security on outbound packets if it is available but does not require it on inbound packets. Level 2 requires security both outbound and inbound. Level 3 is the same as level 2 except that outbound packets use a security association unique to this socket. A planned enhancement is to also permit an application to request that its session be provided with a new security association to replace the one in use. We consider our new security-related socket options experimental and may alter them somewhat as we gain more experience with application issues.

Our kernel implementation permits a system administrator to define a default or minimum level of security. The default security will be used for all sessions provided with a valid Security Association. Applications may also request security services via the above sockets extensions. The system security is configured using the same matrix of 3 protocols and 4 security levels that we described earlier for use in socket-requested security. We plan to enhance the flexibility of our security policy engine in the future so that the system administrator can have more sophisticated policies than are currently supported.

6.2 Key Management Socket

We also have defined a new protocol family, called **PF_KEY**, for the Sockets application programming interface. This extension to Sockets provides a generic interface between security association management applications, such as a Photuris daemon [KS95], and the kernel's network security data structures.[PAM95] This new generic key management interface is modeled upon the existing routing socket, **PF_ROUTE**.[SkI91] This enhancement permits the key management system to be completely decoupled from the IP security implementation. Multiple key management schemes can be supported concurrently if desired. It also will make it easy to change from one key management algorithm or protocol to a new key management algorithm or protocol. To make such a change, only a new daemon needs to be installed; no kernel modifications or kernel rebuilding is necessary. Many published key management protocols have had flaws discovered years after initial publication[NS78][DS81]. Hence it is important to be able to easily change the key management protocol being used by the system. Our alpha release

includes an application, **key(8)**, that can be used by the system administrator to manage keys and security associations in the kernel. Any key management scheme, whether automatic key management such as Photuris or manual key management such as **key(8)**, can use the **PF_KEY** interface.

6.3 An Example Application: telnet

Most applications will need a small amount of modification to take advantage of IPv6 and its unique features. Even with these modifications, the applications will continue to support IPv4. Most of these modifications are in the socket code, allowing the use of the new **AF_INET6** address family, new data structures, and the corresponding network functions.

We have modified several applications to use IPv6. We describe the modifications required for telnet in the following paragraphs. The telnet application was also enhanced to add command-line options to set the socket security level.⁷

The telnet client first parses the command line and options. If the user has requested IP security services, then the appropriate socket options are set using **setsockopt()**. Telnet then uses the new **hostname2addr()** and **ascii2addr()** functions to seek an IPv6 address for the specified hostname or text representation of an address. If an IPv6 address is returned, telnet then opens a **PF_INET6** socket and begins communicating. The requested security services are automatically applied by the IP security implementation inside the kernel. If an IP security processing error (for example, no security association can be found and one is needed) occurs, then the **EIPSEC** error will be returned to telnet so the user can be informed of the problem.

The IPv4 library functions **inet_ntoa()**, **inet_aton()**, **gethostbyname()**, and **gethostbyaddr()** have been superceded by the new library functions **addr2ascii()**, **ascii2addr()**, **hostname2addr()**, and **addr2hostname()** [GTB95].⁸ These new library functions work equally well for both IPv4 and IPv6, making it easier for applications to support both IPv4 and also IPv6.

In the future, we plan to add a privileged socket option to permit applications that need to bypass IP security to do so (for example, a Photuris daemon). This socket option would fail if the effective user-id of the process connected to the socket was not equal

⁷ Although 4.4 BSD's telnet includes an encryption option, a fatal implementation flaw limits its practical value.

⁸ These new functions were originally suggested by Craig Partridge in an email note to the IETF's IPng mailing list.

to 0 so that ordinary user applications could not bypass system security. Such bypass is needed by key management applications so that they can create the initial security associations. Certain other applications having application-layer security, for example a secured Domain Name Service daemon, might also need to bypass IP security services. Although this has not been implemented yet, we believe it will be straight forward to implement and have already put some of the hooks in place.

7 Performance

Throughput and round-trip latency were measured using Rick Jones' NetPerf tool.[Jon95] NetPerf has more accuracy and reproducibility than some older tools.[Jef95] Except for Table 5, these measurements are for traffic that is neither authenticated nor encrypted, though the security policy checks are still performed.

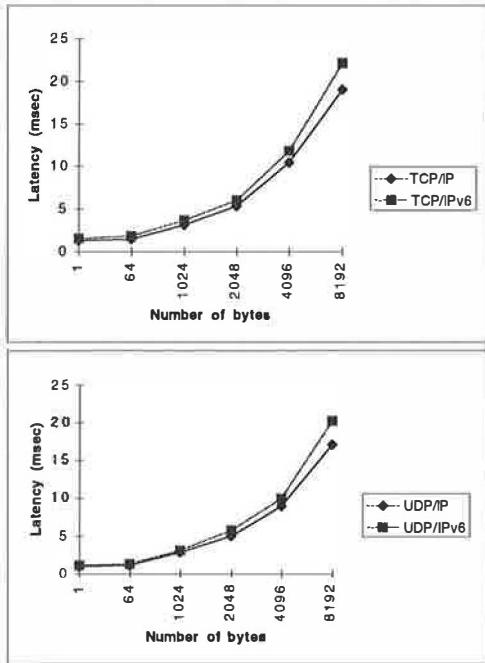


Figure 8: UDP and TCP Latency Graphs

In our alpha release, IPv6 performance is somewhat worse than IPv4. UDP latency, shown in Table 2, and TCP latency, shown in Table 1, both increased for IPv6. The increased latency, shown in Figure 8 is in both the inbound and outbound protocol processing. Comparing longer addresses (four 32-bit words vs. a single 32-bit word) and preparsing of optional headers are the major contributors to the increased latency. We plan to add a fast path bypass

Number of bytes.	IPv4. (msec).	IPv6. (msec).	Percent increase.
1	1.27	1.54	+21%
64	1.45	1.83	+26%
1024	3.12	3.62	+16%
2048	5.34	6.01	+12%
4096	10.4	11.9	+14%
8192	19.0	22.1	+16%

Table 1: TCP Latency

Number of bytes.	IPv4. (msec).	IPv6. (msec).	Percent increase.
1	0.93	1.08	+17%
64	1.13	1.30	+15%
1024	2.82	3.06	+8%
2048	5.00	5.77	+15%
4096	8.89	9.90	+11%
8192	17.0	20.2	+19%

Table 2: UDP Latency

around the parsing code in the future. The lower IPv6 throughput, shown in Table 3 and Table 4, is due to increased latency and larger packet size.

The 4.4-Lite BSD implementation of TCP/IPv4 has had years of optimisation whilst our alpha release has had no optimisation. We believe that an optimised IPv6 implementation will perform at least as well as a similarly optimised IPv4 implementation.

Data size	Socket buffer size	IPv4 (KB/sec)	IPv6 (KB/sec)	Perf. drop
4096	57344	780	731	6.26%
8192	57344	778	729	6.28%
32768	57344	776	730	5.97%
4096	32768	807	763	5.45%
8192	32768	806	758	5.91%
32768	32768	811	762	6.02%
4096	8192	861	775	9.93%
8192	8192	858	784	8.68%
32768	8192	863	784	9.19%

Table 3: TCP Throughput

NetPerf has not yet been modified to use the security socket options. Making such modifications to NetPerf does not appear trivial. The older *ttcp(8)* testing tool was easily modified to use the security socket options. Table 5 indicates throughput differences (measured with *ttcp(8)*) using authentication, transport-mode encryption, and both, versus no security at all. While we have less confidence in the

Data size	Socket buffer size	IPv4 (KB/sec)	IPv6 (KB/sec)	Perf. drop
64	32767	537	500	6.82%
1024	32767	1144	1125	1.60%

Table 4: UDP Throughput

absolute values for *ttcp(8)* than for NetPerf, we believe the relative performance degradation shown by *ttcp(8)* is meaningful. Our security implementations have not been optimised at all. We believe that we can noticeably improve our encryption performance by encrypting and decrypting in place and removing memory copies. Hardware implementations of DES that run at 1 Gbps exist.[Sch94] Implementations seeking high performance should probably use such encryption hardware.

Security Features	Throughput (KB/sec)
None	~775
Authentication	~345
Encryption	~192
Both	~153

Table 5: Impact of IPv6 Security On Throughput.

8 Summary

This paper has described a freely distributable prototype implementation of IPv6 based on 4.4 BSD-Lite. There are a number of implementation differences between IPv4 and IPv6 due to packet format differences and also protocol differences. Some of the assumptions made and techniques used by the IPv4 implementation are no longer valid for IPv6. Because the implementation includes the cryptographic security mechanisms mandatory for IPv6, any networked application can now have the security it desires without having to implement it at the application layer. Performance of TCP/IPv4 and TCP/IPv6 has been compared.

9 Acknowledgments

This work has been funded by the Information Security Program Office (PD71E) of the US Space & Naval Warfare Systems Command since 1992 and also by the Computer Systems Technology Office of the Advanced Research Projects Agency (ARPA/CSTO) since 1995. We are grateful for their support.

References

- [Atk95a] Randall Atkinson. IP Authentication Header, August 1995. RFC-1826.
- [Atk95b] Randall Atkinson. IP Encapsulating Security Payload (ESP), August 1995. RFC-1827.
- [Atk95c] Randall Atkinson. IP Security Architecture, August 1995. RFC-1825.
- [DC95] Steve Deering and Alex Conta. ICMP for the Internet Protocol version 6, June 1995. Work in Progress.
- [Dee89] Steve Deering. Host extensions for IP Multicasting, August 1989. RFC-1112.
- [Dee91] Steve Deering. ICMP Router Discovery Messages, September 1991. RFC-1256.
- [Dee93] Stephen E. Deering. SIP: Simple Internet Protocol. *IEEE Networks*, 7(3):16–28, May 1993.
- [DH95] Steve Deering and Bob Hinden. IPv6 specification, June 1995. Work in Progress.
- [DS81] D.E. Denning and G.M. Sacco. Timestamps in key distribution protocols. *Communications of the ACM*, 24(8):533–536, August 1981.
- [FMMT84] R. Finlayson, T. Mann, J. Mogul, and M. Theimer. Reverse address resolution protocol, June 1984. RFC-903.
- [GN95] Robert E. Gilligan and Erik Nordmark. Transition Mechanisms for IPv6 Hosts and Routers, May 1995. Work in Progress.
- [GTB95] Robert Gilligan, Susan Thomson, and Jim Bound. IPv6 Program Interfaces for BSD Systems, July 1995. Work in progress.
- [Hin94] Robert Hinden. Simple Internet Protocol Plus white paper, October 1994. RFC-1710.
- [Jef95] Jeffrey D. Chung and C. Brendan and S. Traw and Jonathan M. Smith. Event-Signaling within Higher Performance Network Subsystems. In *Proceedings, High Performance Communications Subsystems*, Mystic, CT, August 1995.

- [Jon95] Rick A. Jones. NetPerf: A Network Performance Benchmark (Revision 2.0), February 1995. Technical Report.
- [KS95] Phil Karn and William Simpson. The Photuris Session Key Management Protocol, October 1995. work in progress.
- [LM91] X. Lai and J. Massey. A Proposal for a New Block Encryption Standard. In *Advances in Cryptology – EUROCRYPT ’90 Proceedings*, pages 389–404, Berlin, 1991. Springer-Verlag.
- [LMQK89] Samuel J. Leffler, Marshall Kirk McKusick, Michael J. Karels, and John S. Quarterman. *The Design and Implementation of the 4.3 BSD UNIX Operating System*. Addison-Wesley, New York, NY, 1989.
- [Lot92] Mark Lottor. Internet Growth (1981-1991), January 1992. RFC-1296.
- [MD90] Jeff Mogul and Steve Deering. Path MTU Discovery, November 1990. RFC-1191.
- [MKS95a] Perry Metzger, Phil Karn, and William Simpson. The ESP DES-CBC transform, August 1995. RFC-1829.
- [MKS95b] Perry Metzger, Phil Karn, and William Simpson. IP Authentication using Keyed MD5, August 1995. RFC-1828.
- [NNS95] Erik Nordmark, Thomas Narten, and William Simpson. Neighbor Discovery for IP Version 6, September 1995. Work in Progress.
- [NS78] R.M. Needham and M.D. Schroeder. Using Encryption for Authentication in Large Networks of Computers. *Communications of the ACM*, 21(12):993–999, December 1978.
- [PAM95] Bao G. Phan, Randall J. Atkinson, and Daniel L. McDonald. PF_KEY: Key Management Support inside 4.4 BSD Unix, December 1995. Technical Report.
- [Plu82] D. Plummer. Ethernet address resolution protocol, November 1982. RFC-826.
- [Pos81] Jon Postel. Internet Control Message Protocol, September 1981. RFC-792.
- [RLH⁺95] Yakov Rekhter, Peter Lothberg, Robert Hinden, Steve Deering, and Jon Postel. An IPv6 Provider-Based Unicast Address Format, August 1995. Work in Progress.
- [Sch94] Bruce Schneier. *Applied Cryptography*. John Wiley & Sons, New York, NY, 1994.
- [Skl91] Keith Sklower. A Tree-Based Packet Routing Table for Berkeley UNIX. In *Proceedings of the Winter ’91 USENIX Conference*, Dallas, TX, January 1991. USENIX Association.
- [TN95] Susan Thomson and Thomas Narten. IPv6 Stateless Address Autoconfiguration, October 1995. Work in Progress.
- [ZBE⁺93] L. Zhang, R. Braden, D. Estrin, S. Shenker, and D. Zappala. RSVP: A New Resource ReSerVation Protocol. *IEEE Networks*, September 1993.

Supporting Mobility in MosquitoNet

Mary G. Baker, Xinhua Zhao, Stuart Cheshire, Jonathan Stone
Stanford University

Abstract

The goal of the MosquitoNet project is to provide continuous Internet connectivity to mobile hosts. Mobile hosts must be able to take advantage of the best network connectivity available in any location, whether wired or wireless. We have implemented a mobile IP system that supports seamless switching between different networks and communication devices. In contrast to previous approaches to mobile IP, we believe mobile hosts should not assume any explicit mobility support from the networks they visit, aside from basic Internet connectivity. This decision places extra responsibilities on the mobile hosts themselves. In this paper, we describe the design and implications of such a system. Measurements of our implementation show that the inherent overhead to switch networks (below 10ms) is insignificant compared to the time required to bring up a new communication device.

1. Introduction

We envision that ubiquitous network connectivity will someday be a reality. In the future, the Internet will be a collection of different services, both wired and wireless, often with overlapping areas of coverage. Through the combination of these services, especially with the rapid growth of wireless networks, it will almost always be possible for a mobile host to remain connected to the Internet, or at least to reconnect when so desired.

Based on our vision of this future global internetwork, the MosquitoNet project has been working on supporting continuous (or seemingly continuous) connectivity. By continuous connectivity, we mean that a mobile host can send and receive packets whenever it wishes, though the available quality of network service may vary widely.

We believe continuous connectivity is not only feasible but also crucial to make the most out of portable computers. More personal computer users are using their computers mainly for communication. It is also

clear that consumers want continuous connectivity available to them, as shown by the recent introduction of two-way pagers and "500 number" phone numbers that allow consumers to receive telephone calls wherever they go, without informing callers of any change in phone number.

While the physical infrastructure for ubiquitous network connectivity will be available, there are several problems mobile hosts must overcome to make full use of it. This paper addresses two of these problems. The first is that mobile hosts must be able to switch seamlessly between different types of network devices, and the second is that mobile hosts must be able to visit foreign networks that do not provide any support for mobility.

We must be able to switch seamlessly between different network devices to take advantage of whatever connectivity is available. For example, we may need to switch from an Ethernet connection to a radio modem as we leave our offices, taking our computers with us. If we arrive at a site where there is a higher speed connection, we may want to switch once again to take advantage of it, even if the wireless service is still available.

When switching between these networks, it is important to maintain all current network conversations. Restarting all applications every time we change locations is unacceptably annoying. This is especially true for applications that run for extended periods of time and build up nontrivial state, such as remote logins with active processes. As another example, we may have selected a long thread of postings from a newsgroup and wish to read them after we move to a different location. We do not want to restart the news reader and mark the thread again. If we do not support this seamless switching between networks, we would need to rewrite all these applications to save and restore their own states. In MosquitoNet, we make this seamless switching possible without requiring changes to existing applications on mobile hosts or on the hosts corresponding with them.

The second problem we address is how to maintain connectivity when visiting foreign networks that do not explicitly support mobility for visitors. A foreign network is one operated by authorities other than those operating a mobile host's home network. For the foreseeable future, the global network will continue to be a functioning anarchy, i.e., a collection of services under different authorities. For many organizations, there is little motivation to expend much effort solely on the behalf of mobile visitors. For this reason, our mobile hosts do not require any mobility support from the networks they visit. It is this issue of mobility support in foreign networks that distinguishes the emphasis of our work from previous work on providing host mobility.

Even if future networks adopt a standard mobility protocol, our system provides mobility in the current network. It took ten years for IP multicast to reach its current stage of deployment. We do not want to wait another ten years for mobile IP support that assumes the existence of "agents" operating on a mobile host's behalf in every network it visits. MosquitoNet mobile hosts do not require such changes or additions to network infrastructures outside their home domain.

To solve these problems and gain more experience with mobility, we have designed and implemented a system that requires support only in the home domain of the mobile host and on the mobile host itself. While our approach simplifies the system in some ways, it raises design issues for the mobile host's network software. Our software must now support the mobile host's interactions with foreign networks as well as its interactions with its home network. Determining where we can or should keep mobility transparent to the mobile host's networking software is more complex in our system than in systems with foreign network support.

This paper presents our protocol, our resolution of these design issues, and our system's performance. Our measurements show that switching between available networks causes little disruption to applications running on mobile hosts or on the hosts corresponding with them.

The next section describes the differences between our approach and the previous related work. Section 3 presents our mobile IP system design and implementation. Section 4 provides some performance evaluation of the system. Section 5 describes what we have learned as a result of implementing this system, including the advantages and disadvantages of operating without agents in foreign networks, and transpar-

ency issues for mobile IP implementations. Section 6 describes some future work for our project. The final sections give some concluding remarks, together with release information for our software.

2. Comparison with Previous Work

Several systems have been proposed (and some implemented) that provide host mobility in the Internet. Existing Internet routing protocols are used in all these systems. As illustrated in Figure 1, the systems share several components with our approach. A *mobile host* (MH) is a host that can be reached through a constant *home IP address* regardless of its current location. A *correspondent host* (CH) is a host that communicates with a mobile host. The correspondent host could itself be mobile. Another component in common is a stationary host, called a *home agent* (HA). The home agent takes packets from correspondent hosts addressed to a mobile host's home IP address and forwards them to the mobile host's current point-of-attachment. This point-of-attachment in a foreign network is often called the mobile host's *care-of address*. The home agent typically forwards packets to the mobile host by *tunneling* them. This

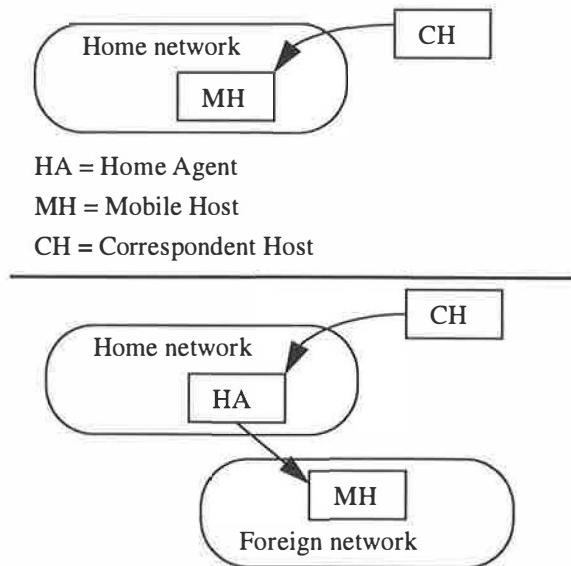


Figure 1. Basic mobile host scenario: The top half of the figure shows a correspondent host communicating with a mobile host that is still on its home network. The bottom half of the figure shows the path packets take when the mobile host moves to a foreign network. The correspondent host continues sending packets to the mobile host's home IP address. An agent in that network (the home agent) takes responsibility for forwarding these packets to the mobile host's new location on the foreign network.

means the home agent encapsulates each packet with an extra IP header that directs the packet to the mobile host's current care-of address. Once received, this packet must be *decapsulated* to strip off the outside header before delivery to an application on the mobile host. In this way the application receives a packet that looks like a normal packet it would receive while on its home network.

The main point that distinguishes our approach from these other systems is how much support is demanded from the foreign networks a mobile host visits. Previous work usually assumes the existence of a *foreign agent* (FA) in each network the mobile host visits. The foreign agent serves as a temporary point-of-attachment for any mobile hosts visiting its network. This means that the mobile host's home agent tunnels packets to the foreign agent, which then decapsulates them and hands the original packets directly to the mobile host on its network. The IP address of the foreign agent becomes the care-of address for the mobile host.

In contrast, our approach only requires of the host network its ability to provide a dynamically-assigned temporary IP care-of address for the mobile host itself. This IP address could be assigned by hand, but this functionality is more easily provided automatically by DHCP [4] or other link-level address negotiations such as those used by PPP and SLIP services. Without a foreign agent, networking software in the mobile host decapsulates the tunneled packets. In effect, we have collocated a simple foreign agent on the mobile host itself. The difference in the assignment of care-of addresses between our design and designs that use foreign agents is further illustrated in Figure 2. We describe the advantages and disadvantages of leaving out the foreign agent in more detail in Section 5.1.

Below we compare some previous mobile IP systems to our work.

The Columbia system [6] takes the so-called 'embedded network' approach. The approach requires a special kind of router, called a Mobile Support Router. The Mobile Support Routers work closely together to make the partitioned physical networks (called cells) appear as a single subnet. While it is optimized for localized mobility, it is hard to scale beyond the scope of a single organization such as a university.

The IMHP (Internet Mobile Host Protocol) [13] and the Harvard system [2] are roughly the same, though they were developed independently. The strong point

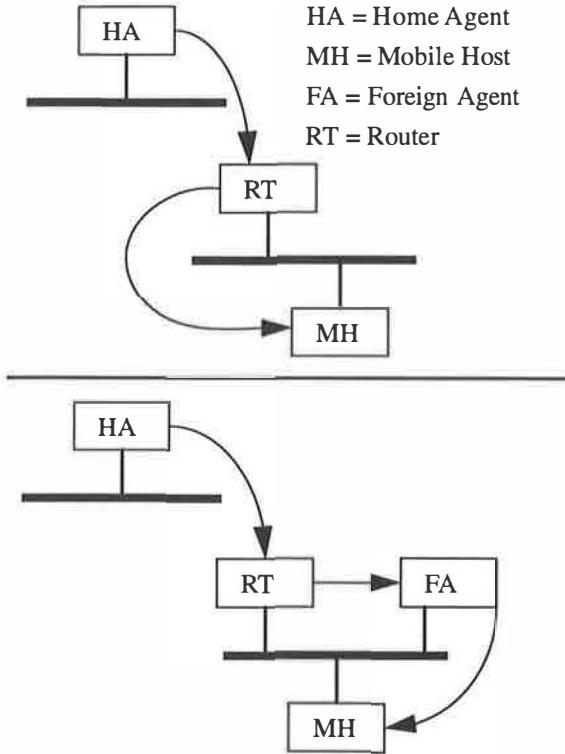


Figure 2. Care-of addresses: The top of the figure shows the home agent forwarding a packet to the mobile host on a network without a foreign agent. The destination address, or care-of address, for the packet is a temporary IP address on the foreign network. The router contains a map entry from this IP address to the hardware address of the mobile host's interface. The bottom of the figure shows a home agent forwarding a packet to a mobile host on a foreign network with a foreign agent. In this case, the mobile host's care-of address is the IP address of the foreign agent. The router hands the packet to the foreign agent, which then delivers it to the mobile host. In this case, the foreign agent contains a map entry that translates from the mobile host's home IP address to its hardware address.

of the proposals is that they can provide host mobility over a wide area and converge to an optimal route efficiently. But to obtain optimal routing, the MH's previous FA, its CHs, and the routers along the way (called cache agents) are used to cache the location binding for the MH. The problem with this is that it requires adding this extra support to many entities in the Internet.

VIP [16] places even more requirements on the existing infrastructure of the Internet. Its key point is to separate logical identifiers from physical identifiers. The network layer is divided into two sublayers: a Virtual Network Sublayer and a Physical Network Sublayer. The virtual to physical mapping information of migrating hosts is cached in Address Mapping

Tables on source hosts or intermediate routers for address resolution. Entries in these tables are updated (created or invalidated) by control packets. This technique can be applied to other identifiers, such as group identifiers for multicast communications, making this a general mechanism. But this approach also has the problem of requiring many changes to existing routers.

The current draft of the IETF Mobile IP Working Group [10] is similar to the IMHP and the Harvard system, but it differs from them in that it treats the support for optimal routing as an extension. In the proposal, the home agent has primary responsibility for processing and coordinating mobility services, while the foreign agent only has a passive and minimal role. Although a full foreign agent is expected to do more, the protocol only requires it to relay registration requests (change-of-location notifications) from the mobile host to its home agent and decapsulate packets for delivery to the mobile host. The protocol does not require any extra code on systems other than the mobile hosts, home agents and foreign agents.

We chose to base our implementation on the IETF specification, because it entails the fewest unrealistic expectations about the amount of support required from other hosts and routers in the Internet. However, we have reduced these expectations even further by leaving the foreign agent out of our basic protocol. While the most recent draft of the IETF specification suggests that the mobile host could use a dynamically acquired IP address instead of the foreign agent as its care-of address, it does not discuss the design implications of this approach, and it still encourages the use of full foreign agents.

Several of the above systems include security mechanisms. We believe that strong security is as necessary for mobile IP as it is for all networking software, but a full discussion of mobile IP security issues is beyond the scope of this paper. We do not yet implement any special security measures in our system. Although many people worry that mobile computing poses special security risks, the majority of the perceived problems are existing problems of the entire Internet that are simply brought into much sharper focus by the advent of mobile hosts.

3. MosquitoNet Mobile IP Design

In this section we describe the current design of our mobile IP system. We start by describing the roles of the mobile host, home agent and correspondent hosts

and how they implement our basic mobile IP protocol. We then list possible optimizations to this basic protocol, including a simple one we have implemented. Finally, we describe the structure of our software. The software must correctly handle the basic protocol without precluding the use of the optimizations. We have implemented this design in the Linux operating system, version 1.2.13.

3.1 System Components

Of the three basic entities in our mobile IP system, the mobile host, home agent, and correspondent hosts, only the mobile host and home agent require mobility support. The mobile hosts require somewhat more support in our system than in implementations with foreign agents, since our mobile hosts must be able to encapsulate and decapsulate packets on their own. We consider this reasonable, because we have control over the software on mobile hosts.

The mobile host must be able to receive packets from correspondent hosts wherever it moves. To remain reachable, it must receive packets addressed to it at its home network. When at home, it directly receives these packets. When it leaves and connects to another network, these packets must be forwarded to it. To accomplish this, the mobile host needs to acquire a temporary care-of IP address from the new network (perhaps dynamically via DHCP). Since our approach does not assume the existence of a separate foreign agent in the new network, the mobile host serves as its own foreign agent and sends a registration message to its home agent to notify it of the new care-of address. At this point the home agent is prepared to tunnel any packets it receives from a mobile host's correspondent hosts to the mobile host's current care-of address, as previously illustrated in Figure 1.

The mobile host must also be able to send packets as well as receive them. At home, it sends packets in the normal fashion. While away from home, in our basic protocol, outgoing packets from the mobile host are also tunneled through the home agent to the correspondent hosts. With no foreign agent on the foreign network, the mobile host must encapsulate these outgoing packets itself. We can sometimes optimize the route for outgoing packets by sending them directly to the correspondent hosts, as described in Section 3.2.

The basic role of a home agent is two-fold. It must decapsulate packets sent from the mobile host for delivery to correspondent hosts, and it must encapsulate packets sent from correspondent hosts for delivery to the mobile host's care-of address. To

encapsulate packets sent to the mobile host, the home agent must be able to intercept them when they arrive in the home network. To intercept these packets, the home agent must function as the ARP proxy for the mobile host upon receiving its registration request. This is done by adding an ARP entry in the home agent's own ARP cache. The home agent must then broadcast a gratuitous ARP on behalf of the mobile host to void any stale ARP cache entries on hosts in the same subnet as the mobile host's home. The home agent also adds an entry to its route table specifying that all packets for the mobile host's home IP address must be encapsulated. It adds a *mobility binding* to an internal table to record the mobile host's care-of address and other information such as the lifetime of the registration and any authentication information.

When the mobile host returns home, it de-registers with the home agent, which then removes the mobility binding and the special route table entry. The home agent should also stop functioning as the ARP proxy for the mobile host.

3.2 Routing Optimizations

Our basic mobile IP protocol uses a simple model of mobile networking in which outgoing and incoming packets are delivered indirectly via the home agent, using an encapsulating tunnel, to make the mobile host appear as if it were still on its home network. While this basic protocol is simple and always works, the extra path through the home agent adds latency to packet delivery. This section lists some desirable routing optimizations for outgoing packets from the mobile host, showing how they can be performed in a system without foreign agents. (We do not consider routing optimizations for the reverse path – from correspondents to the mobile host – as they are necessarily more difficult and we have not yet implemented any of them. These optimizations require the correspondent host to be able to locate the mobile host at its care-of address.)

Optimizations for packets originating from the mobile host can be evaluated based on at least three criteria: First, does the optimization improve the route a packet takes, or does it eliminate the overhead of encapsulation? (Encapsulation adds 20 bytes or more to the packet length and requires extra processing.) Second, does the optimization require some understanding of mobility on the correspondent hosts? Some correspondent hosts may be mobile themselves or may run mobile-aware software. We call these *smart correspondent hosts*, and we'd like to take

advantage of them when possible. A third criterion is whether routers or firewalls are likely to object to the way a packet is addressed or sent.

To improve both the path packets take as well as eliminate encapsulation overhead, a mobile host can send packets directly to its correspondent host. This forms a *triangle route*, as illustrated in Figure 3. For this simple optimization, we set the source IP address of the packets from the mobile host to the mobile host's home IP address rather than its current care-of address. In this way, the mobile host can move as many times as it desires without the correspondent host noticing. As far as the correspondent host knows, the mobile host is always at its home address. If the source address were allowed to reflect the current care-of address, then packets with a changed address would not be recognized as coming from the mobile host without modifications to the correspondent host's software.

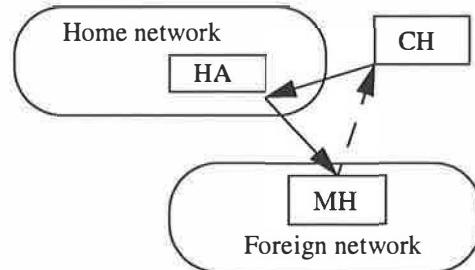


Figure 3. Triangle route optimization: This figure shows a simple route optimization called a triangle route. The optimization allows the mobile host to send a packet to the correspondent host without tunneling it through the home agent. This path is shown with a dotted line. The path for packets from the correspondent host to the mobile host remains unoptimized and passes through the home agent.

The problem with this optimization is that it does not work with some security-conscious routers that forbid transit traffic. Transit traffic is traffic with a source address not local to the network, as is the case with a packet using the mobile host's home IP address as source address. If the foreign network has been set up to forbid transit traffic, then the routers will drop outgoing packets from a mobile host using the triangle route optimization. This problem does not occur with the unoptimized route, which is why we have implemented both options. If we find that we cannot use the optimization, through failed attempts to "ping" a correspondent host, then we can revert to using the unoptimized route. We can cache this information for further use in the Mobile Policy Table described in Section 3.3.

A variant of the triangle route optimization, suitable for use on networks that forbid transit traffic, still sends the packet directly to the correspondent host but encapsulates the packet using the mobile host's local source IP address. Now the packet will not be dropped by the filters in the local router, because it has a valid local source address. This optimization eliminates the sub-optimal routing for outgoing packets but not the encapsulation overhead. It is appropriate when the mobile host knows that the destination host has transparent IP-in-IP decapsulation capability such as is found in recent Linux development kernels.

Finally, there are times when a mobile host wishes to talk directly with other hosts, without any attempts to hide its current physical location. This is the case when answering foreign-network management probes (such as ICMP ping and SNMP). This could also occur if the mobile host contacts correspondents for short periods knowing it will not accumulate any connection state with them. For example, the mobile host may request a web page directly from a web server. The web server simply responds and does not need to track the mobile host further. If the mobile host moves before receiving the response, the user can retry the operation. For this optimization the mobile host sends packets directly to the correspondent host, without encapsulation and without setting the source IP address to its home IP address. This is the most efficient mechanism, but it provides no mobility support. Unless the correspondent host has extra mobility support itself, it will not be able to continue communicating with the mobile host if it moves again.

Given these optimizations, a mobile host must make three decisions about how to send a packet: 1) whether to send a packet directly or tunnel it through the home agent, 2) if sending the packet directly, whether to encapsulate it, and 3) whether to use its home IP address or local address as the source address of the packet. While we only implement the triangle route optimization, it is important that our software structure not preclude these other optimizations, as described in the following section.

3.3 Software on the Mobile Host

Practicality dictates that we write our mobile IP code in such a way as to minimize the impact on the rest of the existing Linux kernel code. In fact, our only changes to the kernel network software are to add mobility support to IP by 1) altering the route lookup function `ip_rt_route()`, 2) adding a Mobile Policy Table, and 3) adding a virtual link-level interface,

called VIF, to encapsulate packets. In this way we are able to implement both our basic protocol, allow for the previously described optimizations, and function without foreign agents.

Figure 4 illustrates the organization of our software when sending out a packet from the mobile host. The transport level protocols deliver a packet to IP, which we have extended for mobility support; our modified `ip_rt_route()` uses its Mobile Policy table combined with the usual routing table lookup to determine how the packet should be treated. The source and destination addresses of the packet are parameters to these table and function lookups. As a result, the packet may be treated as one of two basic types:

- Outside the scope of mobile IP - Some applications and services set the source address of a packet to a specific outgoing network interface, and we do not interfere with their intentions. For instance, an application may use the local-loopback interface, and there is no reason to send such packets through the home agent. Other mobile-aware applications will have their own reasons for specifying particular network interfaces. If the source address of the packet is already set, IP sends it directly to the appropriate interface, as would be done without mobile IP.
- Requiring mobile IP - If the source address of the packet is unspecified, then we must assume that the application that generated it is not mobile-aware. In this case we set its source address to the home IP address. If the application has already set the source address to the home IP address, this too means the packet is subject to mobile IP.

If we decide the packet should use mobile IP, our next decision is whether we should use any optimizations. The decision is based upon the destination address and information stored in the Mobile Policy Table. If we determine a correspondent host has extra mobility support or that a route optimization is appropriate, we cache that information in the table. In our current implementation, though, the only choices are whether to use the triangle route or to tunnel the packet through the home agent. Also, we do not yet update the table dynamically. Using the triangle route, we send the packet directly out the appropriate interface. If the packet requires encapsulation, we send it to our new virtual interface, VIF, for encapsulation.

Upon receiving a packet, VIF adds the extra IP header and sets the appropriate source and destination address in the outer (encapsulating) header. It then hands the packet back to IP for delivery to the appro-

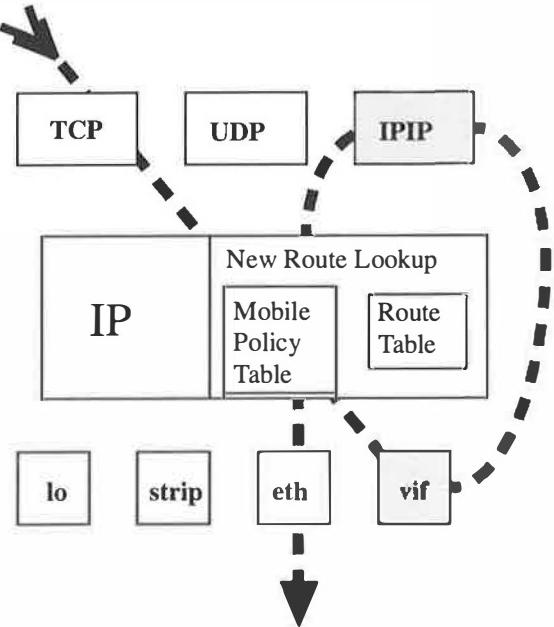


Figure 4. Outgoing packet on a mobile host: This figure shows the link, network, and transport layers of our protocols. The bottom (link) layer shows the device interfaces, with vif being a virtual interface that accepts packets requiring IP-within-IP encapsulation. This layer also includes the loopback interface (lo), the ethernet interface (eth), and our wireless radio interface (strip). The network layer uses the IP protocol. Besides TCP and UDP, we have added the IP-within-IP processing module (IPIP) to the transport layer. The shaded boxes indicate that vif and IP-within-IP are actually implemented as one module for efficiency. The wide dashed line shows the path an outgoing TCP packet might take in the basic mobile IP protocol.

priate physical interface. IP again looks at the packet addresses and makes its decisions, as above. To ensure the packet doesn't get encapsulated again, VIF must set the source address in the outer (encapsulating) header to a specific physical interface. In this sense, we can consider IP-within-IP (IPIP) to have delivered a new packet to IP, which treats the packet based on the same set of rules as before.

To keep the implementation simple, we have separated out routing decisions and mobility decisions. This allows us to leave the routing tables unchanged and merely add our Mobile Policy Table for IP's use. In this way, we are able to make these policy decisions by altering only a single kernel routine: the kernel's IP route lookup function, `ip_rt_route()`. This function returns, for any given destination address, both the recommended interface to use to reach that destination and the recommended source address to use. By overriding this routine, we are able to give appropriate responses to IP, TCP, and any other current or future software that may call

`ip_rt_route()`. If a policy decision indicates the packet should have the home IP address as its source address, we merely return this address from the function call. If a policy decision indicates that we should encapsulate the packet, we return the encapsulating interface as the recommended interface to use. Linux's existing `ip_rt_route()` routine uses the kernel's routing tables to determine its answer; our enhanced routine additionally consults the Mobile Policy table and uses information from both tables to determine its response.

So far we have described the mechanism for sending a packet, but the mobile host must also process received packets. This is simpler than sending packets; because there are no policy or routing decisions to make, all necessary information for processing the packet is contained in its headers. The packet arrives at a physical interface and is delivered to IP. If the packet is an IP-within-IP packet, it will be decapsulated and will take the reverse of the dotted path shown in Figure 4.

3.4 Software on the Home Agent

The home agent shares with the mobile host the need for a virtual interface for encapsulation. This is because the home agent must encapsulate packets destined for the mobile host and tunnel them to the mobile host's current care-of address. For each mobile host away from home that has registered its current location with the home agent, there is an entry in a mobility binding list on the home agent that keeps track of information about the mobile host, such as its care-of address and the lifetime of its current registration. The home agent also adds an entry in the routing table to indicate that all subsequent packets for the mobile host's home IP address should be sent through the VIF. When the packets are sent to the VIF, they get encapsulated within another IP packet and then tunneled to the mobile host's current care-of address.

Another function of the home agent is to receive encapsulated packets from the mobile host and forward them to the correspondent host. This only involves decapsulation and IP forwarding. The decapsulation software is the same as that in the mobile host, and we simply turn on IP forwarding in the Linux kernel. In fact, more recent development versions of the Linux kernel (1.3 and later) include a decapsulation module. In these versions, we will not need to include our own.

4. Performance

This section describes our test-bed and contains performance data on the cost of a hand-off when a mobile host switches between different networks. Our test-bed contains both wired and wireless networks so that we can test switching between different types of device interfaces. The goal is that our wireless devices will provide a constantly available network connection, while the wired devices will provide a faster connection where available.

Our mobile hosts are Gateway 2000 Handbook486s running Linux [8]. The Handbooks are 40 Mhz sub-notebooks that weigh less than three pounds and are thus very comfortable to carry around. Each Handbook has a PCMCIA card slot that we use for Ethernet connectivity with a Linksys Ethernet card. Each Handbook also has a 115.2 Kbit/second serial port that we use to connect to our wireless devices.

Our wireless devices are Metricom radio modems [14]. While these devices are commonly used to emulate a Hayes modem for point-to-point connections, Metricom also provides a connectionless datagram mode, called "Starmode," that enables a radio to send packets to any number of other radios individually. We have written a Linux driver (STRIP) that allows us to run IP over the radios' Starmode. In theory, Metricom radios can send 100 Kbits/second through the air, but in practice 30-40 Kbits/second is the best we achieve [3].

We use a Pentium 90 as the router to the home network of our mobile hosts. It is also usually used as the home agent for the mobile hosts. However, our implementation does not require the home agent to be collocated with the router; rather, we only require the home agent to be one of the hosts on the same network.

Figure 5 shows the setup of our test environment. Net 36.135 is a wired (Ethernet) subnet for our research group, serving as the home network for the mobile hosts. Net 36.8 is a wired (Ethernet) subnet belonging to the Computer Science Department and connected to the Internet. Net 36.134 is a wireless subnet for our Metricom radio devices. The results we report below are for a correspondent host located on net 36.8, but we received similar results for a correspondent host located on a campus network outside the department.

We performed two types of experiments. The first measures the disruption to the system when switching the care-of IP address of the mobile host to another IP

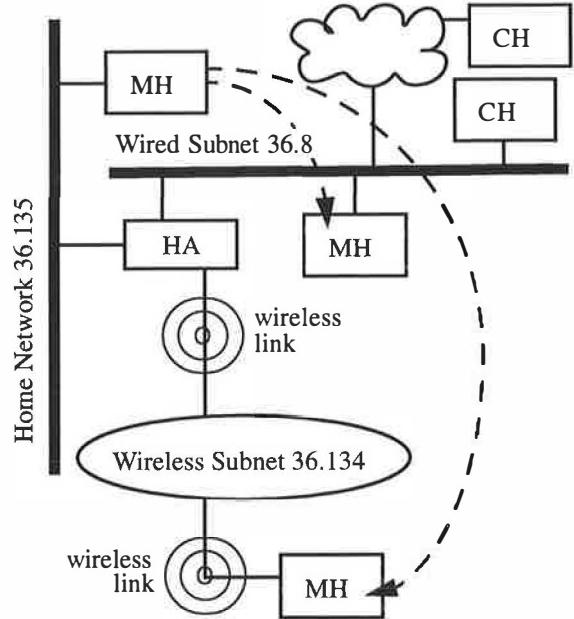


Figure 5. MosquitoNet test-bed: The home network for the mobile host (MH) is net 36.135. As shown by the dashed arrows, the mobile host visits either network 36.8, which it accesses using its Ethernet interface, or net 36.134, which it accesses using its wireless interface. The cloud is our artistically-challenged method for denoting the rest of the Internet. The correspondent host (CH) in this example is on net 36.8 or elsewhere in the Internet. In this figure the home agent (HA) is collocated on the router connecting our networks, but it could instead be on some other host in the home network.

address on the same wired subnet. We measure this disruption in terms of lost packets due to software overhead and address registration. Switching between addresses on the same subnet is not something we usually do in practice, but it is a measurement of the minimal essential software overhead of our system. For these tests, a correspondent host continuously sends a UDP packet to the mobile host every 10 milliseconds, and the mobile host echoes the packet back. We then measure the number of packets that were lost during the interval in which the mobile host switches addresses. This interval occurs between the time the mobile host can no longer accept packets at the old care-of address and the time it registers the new care-of address with its home agent. Packets in flight during this time will be lost by arriving at the old address. No matter how small this interval is, it is always possible for some packet in flight to arrive during this time; however, the larger the interval, the more packets we lose.

Out of the twenty iterations of this experiment, sixteen tests showed no packet loss, and the other four tests lost one packet each. This indicates that the

interval during which packets can be lost is under 10 ms. We could not run the tests with a smaller interval due to the accuracy available from our current measurement tools.

The second experiment measures the disruption when switching between two types of devices, both from wired to wireless and from wireless to wired. We further subdivide this latter experiment to distinguish between *cold switching* and *hot switching*. In cold switching, we shut down one interface before starting up the other. The mobile host deletes the route to the first interface, brings the interface down, brings the new interface up, adds its route, and finally registers the new IP address with its home agent. Bringing an interface up or down usually just involves configuration in software, but some devices may also require hardware interaction. In hot switching, both of the interfaces are available and we just switch from one to the other. The mobile host merely changes its route and registers the new address with its home agent. For these tests the correspondent host sends a UDP packet every 250 milliseconds. We chose the 250ms interval

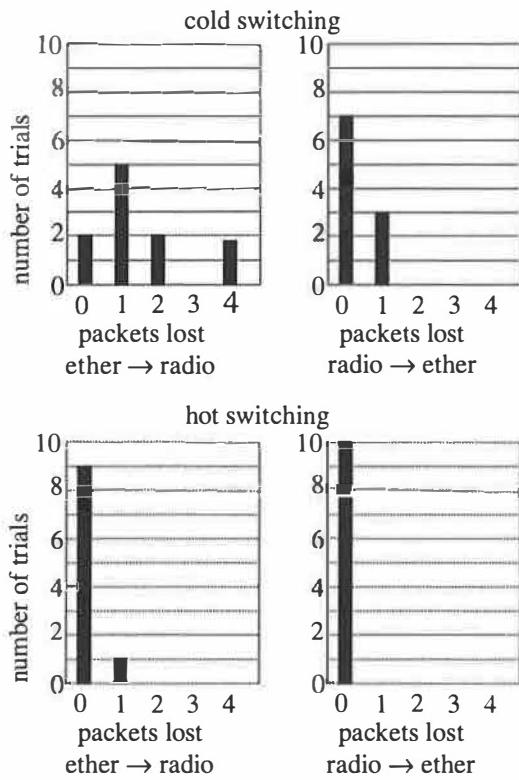


Figure 6. Device switching overhead: This figure shows the results from our experiments measuring packet loss when the mobile host switches between different network interfaces. We repeated each experiment ten times. The height of the bars shows the number of iterations in which the given number of packets were lost.

because the round-trip time between the home agent and the mobile host through the radio interface is 200~250ms. Figure 6 shows our results for this second set of experiments, after running each experiment 10 times.

When doing cold switching between different devices, the period of time during which packets are lost is generally less than 1.25 seconds. The longer time interval is due to bringing up the new interface. This seems acceptable when compared with the time spent physically switching between two devices. When doing hot switching, we usually see no packet loss. (The only lost packet we observed was dropped by the radio itself and was not a result of the device switch.) This is what we would expect, since no packets should be lost if both interfaces are available.

The results for hot switching show that being able to bring up one interface before turning off the other is advantageous. Whether this is possible in practice depends on whether the user or the network monitoring software on the mobile host have any warning that connectivity is about to change. With sufficient warning, for instance, the user or the mobile host can bring up a newly available wireless interface before the old interface is disabled.

We have also collected data to break down the time in each step of the mobile host's switch to a new address and its registration with the home agent, as illustrated in Figure 7. The measurement is performed with the mobile host registering a new IP address on the same Ethernet subnet. The data reflects the average of 10 tests. The total time from start to end, including the time used to configure the interface and change the routing table, is 7.39ms. The latency between the

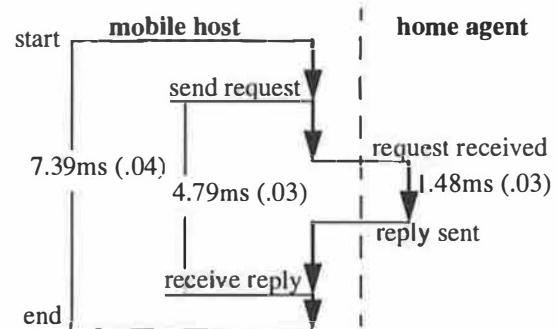


Figure 7. Registration time-line: The graph shows the time the mobile host spends on each step of the registration process. The total time from start to end of the address switch includes the time used in the pre-registration process (configuring the interface and changing the route table) and the post-registration process. The data reflects the average of 10 tests with standard deviations in parentheses.

mobile host sending out a registration request and receiving the reply is 4.79ms. The average time between the home agent receiving the registration request and sending out its reply is 1.48ms. The data shows that the software overhead in the registration process is small, and the home agent should be able to deal with a large number of mobile hosts simultaneously.

5. Designing for Mobility

Over the course of this project, we've dealt with some issues in the design of a mobile network that we believe are likely to arise in other implementations of mobility, especially those that do not use foreign agents. In this section we examine two of these issues. We first list the general ramifications of leaving out foreign agents. Next, we examine in more detail the degree to which we can hide mobility from network software running on the mobile host in the absence of foreign agents. While complete transparency sounds ideal, it is not entirely practical.

5.1 Leaving out the Foreign Agent

Our decision to leave out foreign agents in our basic mobile IP protocol has both advantages and disadvantages. Most of the advantages are related to the resulting reduced dependence on the foreign network. In return, though, we must acquire a temporary IP address in the foreign network, since we cannot use a foreign agent's address as our care-of address. The disadvantages of our decision are related to our use of this temporary IP address.

Advantages:

- + Main advantage: Our mobile hosts can visit networks that do not have a foreign agent.
- + Fault tolerance: The foreign agent is no longer a single point of failure for our mobile hosts' ability to continue communicating with the outside world. While we may be able to fix or restart failed home agents, we probably do not have such control over resources in a foreign network.
- + Scaling issues: We do not need a foreign agent running in every network. We only need home agents running in those networks with mobile clients.
- + Simpler protocol: Leaving out a full implementation of a foreign agent simplifies the essential part of the protocol. However, we must implement a

subset of the foreign agent on the mobile hosts for decapsulating packets and registering with the home agent.

- + Compatibility with IPv6: Although mobility specifications for IPv6 have not been finalized, it appears our approach is likely to be compatible with this protocol. There will be a large IP address space, resulting in less resistance to handing out temporary addresses. Also, it appears there may be support for bindings in routers between static addresses and care-of addresses, further reducing the role of a foreign agent [12].

Disadvantages:

- Main disadvantage: The mobile host needs to acquire a temporary IP address in the foreign network. There may be networks that refuse to do this. (These same networks may also refuse to expend any other resources on visitors, including foreign agents.)
- Security: If packets for a mobile host arrive at a foreign network the mobile host has just left, those packets might be erroneously delivered to a newly arrived host that has been assigned the same temporary address the recently departed host used. This kind of accidental eavesdropping should not happen in practice because a well-written DHCP server would avoid reassigning the same IP address for as long as possible.

Anyone concerned about deliberate malicious eavesdropping should be using end-to-end encryption rather than worrying about address reassignment problems. Packets on an Ethernet or elsewhere on the public Internet can already be easily read by a packet-sniffing program. The only security problem that is truly unique to mobile hosts is the registration of the temporary care-of address with the home agent and with smart correspondent hosts. These registrations should be authenticated with S-key, Kerberos, PGP, or some other similar strong authentication mechanism to protect against denial-of-service attacks in the form of malicious fraudulent registrations.

- Packet loss: Foreign agents may somewhat reduce packet loss. When a mobile host leaves a network, it must inform its home agent of its new care-of address. However, any packets already sent by the home agent before it receives the new registration will arrive at the old network and will be lost. If, however, a foreign agent in the old network

receives the new registration before the packets arrive, it can forward the packets to the mobile host's new care-of address.

The important question, though, is whether the benefit is worth the cost. An important lesson of the Internet is that while it is relatively easy to deliver almost all packets, attempting absolute reliability makes the cost of the system grow towards infinity. Many of the components that make up the Internet exercise the option to drop packets occasionally when the cost of doing otherwise would be unreasonable. Internet protocols do not naively assume perfect delivery, but instead use end-to-end mechanisms to achieve a level of reliability appropriate to their particular needs [15]. Unless further experience with our system dictates that our potentially higher packet loss is a severe handicap, we will stick to our simple implementation.

- More complex mobile host: The lack of foreign agents somewhat complicates the design of the mobile host. This is because our mobile hosts effectively contain a simplified foreign agent. The result is that some networking software on a mobile host must be aware of its actual physical network connectivity and must handle routing operations and changes to the physical network interfaces. This gives us increased flexibility for routing optimizations but presents a new problem for our design: we must understand where mobility can be transparent to the software and where we should expose the physical network. However, this minor increase in complexity is an engineering challenge that is solvable, whereas convincing every independent authority on the Internet to provide foreign agent services would be a political task without end.

Other issues we've found that appear to be distinctions between a foreign agent implementation and our protocol do not seem significant. For instance, route optimizations handled by foreign agents are also possible in our scheme when handled by mobile hosts and home agents, as shown previously.

Despite our desire to make our basic protocol simple and self-contained, there is nothing that prevents us from implementing or using foreign agents. If it becomes appropriate, we can provide extensions to our system to implement foreign agents for visiting mobile hosts that require them. Likewise, we can extend our protocol on mobile hosts so they can take advantage of any foreign agents that happen to exist in networks they visit.

5.2 Design Alternatives

One of the most difficult issues we've tackled in our implementation of mobile IP is how transparent mobility should be to users and software on the mobile host itself. What we've learned through iterative designs is that complete transparency is not flexible enough and is not entirely practical. Our solution is to allow the routing tables to expose the mobile host's physical connectivity to its current network to any software or services that require this. Otherwise, we ensure that all applications on the mobile host and correspondent hosts need not know anything about mobility.

If mobility were entirely transparent to the mobile host, all of its software would have the view that it is always attached to its home network. This sounds ideal but turns out to have unpleasant implications. To hide fully whether the mobile host is at home or abroad, its routing table should always show only one unchanging interface bound to its home IP address. Even while at home this interface could not be associated with any physical network, such as the Ethernet device, since away from home the mobile host might switch to some other device for communication. The home IP address would thus be permanently bound to a virtual interface. The virtual interface would perform encapsulation while away from home and would be similar to our VIF.

The main problem with this approach is that the actual understanding of the physical interfaces would need to be hidden within VIF. This has at least three implications. First, VIF would need to construct its own real routing table showing the state of these physical interfaces. Any routing optimizations would need to be handled entirely within VIF, and this requires exposing information in the socket data structure within the interface since this information is ordinarily accessed before the packet is passed to an interface. Second, applications would not be able to bind source addresses to particular physical interfaces, because they would not be able to see the interfaces. For this same reason, applications would not be able to use two different network services at once, even if they wished to take advantage of their different characteristics for different purposes. Third, we would need to handle ICMP routing redirects differently. If a mobile host communicates with a correspondent host on the network it is visiting, the mobile host may receive routing redirects for the correspondent host that would ordinarily override any default route.

Our experience indicates that an implementation that gives up some transparency is appropriate. This is because a mobile host visiting a foreign network really has two distinct roles to play. The first is its *home* role, in which it appears virtually connected to its home network and sends packets using mobile IP features to provide transparency. Packets sent by a MH in its home role should be sent with a source address of the MH's permanent home address, perhaps with encapsulation through the virtual interface.

The second role is the mobile host's *local* role, in which it participates as a host connected to the foreign network. Examples of this second role include answering foreign-network management probes (such as ICMP ping and SNMP), and the lease-refresh of the DHCP-assigned temporary care-of address. The mobile host might also join multicast groups via the foreign network, rather than via the home network. Or it might correspond directly with another host entirely ignoring mobile IP. This is especially useful if the home agent is not reachable or has crashed.

Our belief is that a mobile IP implementation must support both roles. Foreign networks are unlikely to let visiting mobile hosts connect if the mobile hosts do not respond to local network management tools. Applications that wish to take advantage of particular network interfaces, or that use more than one interface at a time, are necessarily mobile-aware. In this sense, the packets they send are a part of a mobile host's local role. Our solution provides flexibility by exposing the mobile host's local role through the routing tables while ensuring that all applications on the mobile host and correspondent hosts need not know about the local role.

Part of the need for the mobile host to play its local role is a result of our decision to leave foreign agents out of our basic protocol. When a mobile host uses a distinct foreign agent, the foreign agent is a default router for the mobile host and is essentially the mobile host's only connection to the network. Because there are two separate routing tables – one on the mobile host, which always points to the foreign agent, and one on the foreign agent, containing true topology information – the conflict between mobility routing and "real" routing disappears, and a simple implementation using a VIF is sufficient but less flexible.

6. Future Work

Although we believe our work shows we can make changes in network interfaces and routing transparent

to higher-level software, we may not wish to hide changes in the underlying network performance characteristics. Bandwidth, latency, bit error rates, security, and cost can all differ significantly from one type of network to another. We believe it may be advantageous to inform upper-layer network protocols and some applications of these changes so they can adjust their behaviors accordingly. Part of our future work is to investigate what common functionalities should be built into the kernel to cope with these varying quality-of-service parameters, and what application programming interface best enables applications to specify their interests and receive notification of any relevant network changes. Developing a clean interface for this is a major goal of our further work.

As for further work on mobile IP, we plan to experiment with techniques for determining when to switch between networks, and we plan to test some additional routing optimizations described in this paper.

7. Conclusions

We have implemented a self-contained mechanism that enables hosts to switch between different networks and network devices without losing connectivity. Our system does not assume the existence of separate foreign agents, allowing our mobile hosts to visit networks that do not support any mobile IP protocols. This enables host mobility over a wide range of networks controlled by different authorities and brings us closer to our goal of ubiquitous connectivity. The performance of our implementation shows that we can switch between devices and IP addresses with minimal disruption to the higher levels of software.

8. Source Code Availability

The software for mobile hosts and home agents will be freely available via the Internet. Please use the following URL for details of our project and software release: <http://plastique.stanford.edu/mosquito.html>. We also hope to release our code for DHCP and an extended version of DNS on Linux. Availability of any of this code will also be posted on our web site.

9. Acknowledgments

We gratefully acknowledge our many discussions on mobile computing with Bart Miller. His thoughts and advice were enormously useful. We also benefited greatly from discussions with Steve Deering, Mike

Spreitzer, Marvin Theimer, and Lixia Zhang. John Chapin, Stacey Doerr, Bart Miller, Elliot Poger, Mema Roussopoulos, Diane Tang, and Marvin Theimer provided their thoughtful comments on the paper. We thank Metricom for their generosity in loaning us radios and answering many technical questions. We thank Xerox PARC for providing financial support for Xinhua. This work has also been supported by Stanford's Telecom Center and an NSF Faculty Career Award (contract CCR-9501799).

10. References

1. Mary Baker, "Changing Communication Environments in MosquitoNet." *Proceedings of the IEEE Workshop on Mobile and Computing Systems and Applications*, December 1994.
2. Trevor Blackwell et al., "Secure Short-Cut Routing for Mobile IP." *1994 Summer USENIX*, June 1994.
3. Stuart Cheshire and Mary Baker, "Experiences with a Wireless Network in MosquitoNet." *Proceedings of the IEEE Hot Interconnects III Symposium on High Performance Interconnects*, August 1995. A version of this paper will also appear in *IEEE Micro*.
4. R. Droms, "Dynamic Host Configuration Protocol." *RFC 1541*, October 1993.
5. Joe Hung, Bart Miller and Mary Baker, CS 244B Course Project, Stanford University Computer Science Department, Spring 1995.
6. John Ioannidis and Gerald Q. Maguire Jr., "The Design and Implementation of a Mobile Internetworking Architecture." *1993 Winter USENIX*, Jan. 1993.
7. David B. Johnson, and Charles Perkins, "Route Optimization in Mobile IP." *Network Working Group, Internet Draft (work in progress)*, July 7, 1995.
8. The Linux Journal, P.O. Box 85867, Seattle, Washington, 98145. Editor Phil Hughes, publisher Robert F. Young, 1 (1), March 1994.
9. P. Mockapetris, "Domain Names - Concepts and Facilities." *RFC 1034*, November 1987.
10. C. Perkins, "IP Mobility Support." *Internet Engineering Task Force, Internet Draft (work in progress)*, July 8, 1995.
11. Charles E. Perkins and Tangirala Jagannadh, "DHCP for Mobile Networking with TCP/IP." *IEEE ISCC'95*, Alexandria, June 1995.
12. Charles Perkins, and David B. Johnson, "Mobility Support in IPv6." *IPv6 Working Group, Internet Draft (work in progress)*, July 8, 1995.
13. Charles E. Perkins, Andrew Myles, and David B. Johnson, "The Internet Mobile Host Protocol (IMHP)." *Proceedings of INET' 94*, June 1994.
14. M. Pettus, "Unlicensed Radio Using Spread Spectrum: A Technical Overview." Available from Metricom, Inc., Sept. 27, 1993.
15. J. H. Saltzer, D. P. Reed, and D. D. Clark, "End-to-End arguments in System Design." *ACM Transactions on Computer Systems*, 2 (4), November 1984.
16. Fumio Teraoka, Keisuke Uehara, Hideki Sunahara, and Jun Murai, "VIP: A Protocol Providing Host Mobility." *Communications of the ACM*, Aug. 1994.
17. Mark Weiser, "The Computer for the 21st Century." *Scientific American*, September 1991.

11. Biographical Information

Mary Baker is an assistant professor in the Departments of Computer Science and Electrical Engineering at Stanford University. Her interests include operating systems, distributed systems, and software fault tolerance. She received her Ph.D. in computer science in 1994 from U. C. Berkeley.

Xinhua Zhao is a Ph.D. candidate in the Department of Computer Science at Stanford University. His interests include operating systems and computer networks. Before coming to Stanford, he was a student at the University of Science and Technology in China.

Stuart Cheshire is a Ph.D. candidate in the Department of Computer Science at Stanford University. His interests include networks and operating systems. Cheshire received his first class honours degree from Sidney Sussex College, Cambridge in 1989.

Jonathan Stone is a Ph.D. candidate in the Department of Computer Science at Stanford University. His interests include networking, distributed systems, and operating systems. He received an M.S. with distinction in computer science in 1991 from Victoria University of Wellington, New Zealand.

The authors' email addresses are
{mgbaker, cheshire, jonathan, zhao}@cs.stanford.edu.

Their postal address is Computer Science Department, Stanford University, Stanford, CA 94305.

World-Wide Web Cache Consistency

James Gwertzman, *Microsoft Corporation*

Margo Seltzer, *Harvard University*

Abstract

The bandwidth demands of the World Wide Web continue to grow at a hyper-exponential rate. Given this rocketing growth, caching of web objects as a means to reduce network bandwidth consumption is likely to be a necessity in the very near future. Unfortunately, many Web caches do not satisfactorily maintain cache consistency. This paper presents a survey of contemporary cache consistency mechanisms in use on the Internet today and examines recent research in Web cache consistency. Using trace-driven simulation, we show that a weak cache consistency protocol (the one used in the Alex ftp cache) reduces network bandwidth consumption and server load more than either time-to-live fields or an invalidation protocol and can be tuned to return stale data less than 5% of the time.

1.0 Introduction

Network traffic continues to grow at a hyper-exponential rate while network infrastructure does not. This means that existing networks are plagued by ever increasing utilization demands. One approach to coping with the increasing resource utilization is to cache data at non-server sites. As service providers such as America Online introduce millions of subscribers to an already overburdened networking infrastructure, it is nearly assured that systems must cache Web objects to facilitate acceptable service. Caching can be quite effective at reducing network bandwidth consumption as well as server load. Netscape, a vendor of Web servers, claimed in March of 1995 that a single local proxy server can reduce internetwork demands by up to 65% [1].

This research was supported by the National Science Foundation on grant CCR-9502156.

The value of caching is greatly reduced, however, if cached copies are not updated when the original data change. Cache consistency mechanisms ensure that cached copies of data are eventually updated to reflect changes to the original data. There are several cache consistency mechanisms currently in use on the Internet: time-to-live fields, client polling, and invalidation protocols.

Time-to-live fields are an *a priori* estimate of an object's life time that are used to determine how long cached data remain valid. Each object is assigned a time to live (TTL), such as two days or twelve hours. When the TTL elapses, the data is considered invalid; the next request for the object will cause the object to be requested from its original source. TTLs are very simple to implement in HTTP using the optional "expires" header field specified by the protocol standard [2]. The challenge in supporting TTLs lies in selecting the appropriate time out value. Frequently, the TTL is set to a relatively short interval, so that data may be reloaded unnecessarily, but stale data are rarely returned. TTL fields are most useful for information with a known lifetime, such as online newspapers that change daily.

Client polling is a technique where clients periodically check back with the server to determine if cached objects are still valid. The specific variant of client polling in which we are interested originated with the Alex FTP cache [6] and is based on the assumptions that young files are modified more frequently than old files and that the older a file is the less likely it is to be modified. Adopting these assumptions implies that clients need to poll less frequently for older objects. The particular protocol adopted by the Alex system uses an *update threshold* to determine how frequently to poll the server. The update threshold is expressed as a percentage of the object's age. An object is invalidated when the time since last validation exceeds the update threshold

times the object's age. For example, consider a cached file whose age is one month (30 days) and whose validity was checked yesterday (one day ago). If the update threshold is set to 10%, then the object should be marked invalid after three days ($10\% * 30$ days). Since the object was checked yesterday, requests that occur during the next two days will be satisfied locally, and there will be no communication with the server. After the two days have elapsed, the file will be marked invalid, and the next request for the file will cause the cache to retrieve a new copy of the file.

There are two important points to note with respect to client polling: it is possible that the cache will return stale data (if the data change during the time when the cached copy is considered valid) and it is possible that the cache will invalidate data that are still valid. The latter is a performance issue, but the former means that, like TTL fields, client polling does not support perfect consistency.

Like TTL, client polling can be implemented easily in HTTP today. The "if-modified-since" request header field indicates that the server should only return the requested document if the document has changed since the specified date. Most web proxies today are already using this field.

Invalidation protocols are required when weak consistency is not sufficient; many distributed file systems rely on invalidation protocols to ensure that cached copies never become stale. Invalidation protocols depend on the server keeping track of cached data; each time an item changes the server notifies caches that their copies are no longer valid. One problem with invalidation protocols is that they are often expensive. Servers must keep track of where their objects are currently cached, introducing scalability problems or necessitating hierarchical caching. Invalidation protocols must also deal with unavailable clients as a special case. If a machine with data cached cannot be notified, the server must continue trying to reach it, since the cache will not know to invalidate the object unless it is notified by the server. Finally, invalidation protocols require modifications to the server while the other protocols can all be implemented at the level of the web-proxy.

In this paper, we examine the different approaches to cache consistency. An ideal cache consistency solution will provide a reduction in network bandwidth and server load at very low cost. In the next section, we discuss cache consistency protocols in general and cache consistency as applied to the Web in particular. Section 3 presents our simulation environment and Section 4 our simulation results. In Section 5, we suggest some areas for future research and conclude in Section 6, with the

suggestion that weakly consistent protocols are a good choice for web consistency.

2.0 Related Work

Danzig et al. motivate the need for hierarchical object caches for Web objects on the Internet by examining how strategically located FTP caches affect Internet traffic [9]. They found that FTP traffic across the backbone could be reduced by as much as 42%, simply by caching FTP files at the juncture between the backbone and regional nets. This result inspired the design of the Harvest object cache, which is a hierarchical proxy-cache [7].

Once the need for caching has been established, it is instructive to consider how to maintain consistency among the caches. While there are a number of approaches for maintaining cache consistency in distributed file systems, there has been little work aimed specifically at evaluating cache consistency protocols on the World Wide Web. Blaze explored constructing large-scale hierarchical file systems [5]. While his architecture is similar to the one we posit for the web [10], the systems are sufficiently different that his results cannot be directly applied. In his model clients can also act as servers and can cache files on a long term basis. This is not necessarily true in the web where clients are often personal computers with limited resources.

The Berkeley xFS system [8] suggests a model of cooperative caching that is also similar to the one we propose for the web [10]. However, it relies on clients, not only for long-term caching, but also to retain the master copy of data. Like other distributed file systems (e.g. the Sprite Distributed File System [13], the Andrew File System [11]), it also assumes objects can be changed by any machine while web objects can be modified only on their primary server.

The web is fundamentally different from a distributed file system in its access patterns. The web is currently orders of magnitude larger than any distributed file system. Each item on the web has a single master site from which changes can be made. This suggests that consistency issues may be simpler because conflicting updates should never arise.

The most widely used web cache is the original server distributed by CERN [12]. The CERN server assigns cached objects times to live based on (in order), the "expires" header field, a configurable fraction of the "Last-Modified" header field, and a configurable default expiration time. Cached objects are returned, without further consultation with the server, until they expire, at which point subsequent

requests cause an “If-Modified-Since” request to be issued.

One study compares the performance of the CERN proxy cache to a specially designed lightweight caching server [15]. The lightweight cache has an independent process that periodically examines cached objects to determine if they have become stale. Staleness is determined using both TTLs and invalidation callbacks from cooperating primary servers. Proxy caches are registered with the primary server so that they can receive invalidation notices. If one views the CERN proxy cache as implementing an NFS-like consistency protocol [14], the new server implements an AFS-like protocol. The comparison focuses on the performance differences between the two servers and does not examine the relative behavior of the different consistency protocols, which is the focus of this work.

To date, the only other detailed examination of consistency protocols is a study by Worrell that compared TTL fields to invalidation protocols [16]. He showed that the bandwidth savings for invalidation protocols and TTL fields could be comparable if the TTL were set to approximately seven days. Unfortunately, with a TTL of 7 days, 20% of the requests returned stale data. We believed that a simple, but adaptive scheme, such as the Alex protocol, might achieve comparable bandwidth savings with substantially better stale hit rates, so we obtained the same simulator used in Worrell’s study and adapted it for a more extensive evaluation. In the process of exploring the Alex protocol, we discovered that the original workload in the Worrell study was inconsistent with the workload we observed in server traces. We hypothesized that, by using a more trace-based workload, the simulation results would change significantly.

The original simulation environment consisted of a cache simulator and a collection of file ages gathered over several months for 4,000 files located around the Web. The simulator modeled a hierarchical caching system and provided both a TTL cache consistency protocol and an invalidation protocol. The invalidation protocol was optimized so that upon receipt of an invalidation message, objects were simply marked invalid, but not immediately retrieved. This increased latency on subsequent accesses, but decreased bandwidth consumption if the object was not accessed again. Finally, the simulator used the average and variance of the file ages to generate a uniform, random stream of file accesses.

3.0 Simulation Environment

We began with Worrell’s simulator and modified it in a number of ways. We made two initial modifications to begin the experiments. First, we added the Alex protocol. Then, in order to isolate the effects of cache consistency policy from the effects of hierarchical caching, we flattened the cache hierarchy to model a single cache.

Worrell’s simulation analyzed the Harvest cache’s hierarchical caching. We wished to separate the issues of hierarchical caching and cache consistency, focusing only on the latter. While eliminating the hierarchy changes the amount of invalidation traffic in the study, in most cases it does not affect the relative traffic of the different invalidation schemes. When it does affect the relative traffic, it does so in a manner that favors invalidation protocols.

Figure 1 shows the cases in which our results may be distorted by collapsing the hierarchy. In all cases where the relative performance of invalidation and time-based protocols is different in the hierarchical and collapsed systems, our simulation favors the invalidation protocols, while our results suggest that time-based protocols are more desirable. Therefore, we expect that time-based protocols in a cache hierarchy will perform even better than our results indicate.

Since we flatten the hierarchy, Worrell’s “goodness” metric of *network hops * number of bytes transferred* is no longer a useful measure. We use the number of bytes required to maintain consistency, including invalidation messages, stale data checks, and file data movement. For the remainder of this paper, we will refer to Worrell’s modified simulator with Alex and a flattened hierarchy as the *base simulator*.

The base simulator produced results very similar to those reported by Worrell. Our next step was to optimize the Alex and TTL protocols in a manner similar to the invalidation protocol optimization. When a cached datum expired, instead of immediately requesting a new copy, the items were marked invalid. Upon next reference, we issued an “If-Modified-Since” request to the server. The item was only retransmitted if it had, in fact, changed since the last time it was sent. In this manner, we traded the latency of the query request for the bandwidth savings, i.e. not having to retransmit data when a valid copy existed in the cache. By combining the query with the retransmit request to yield a “send this file if it has changed since a specific date” request, we avoided extra overhead and still saved bandwidth where

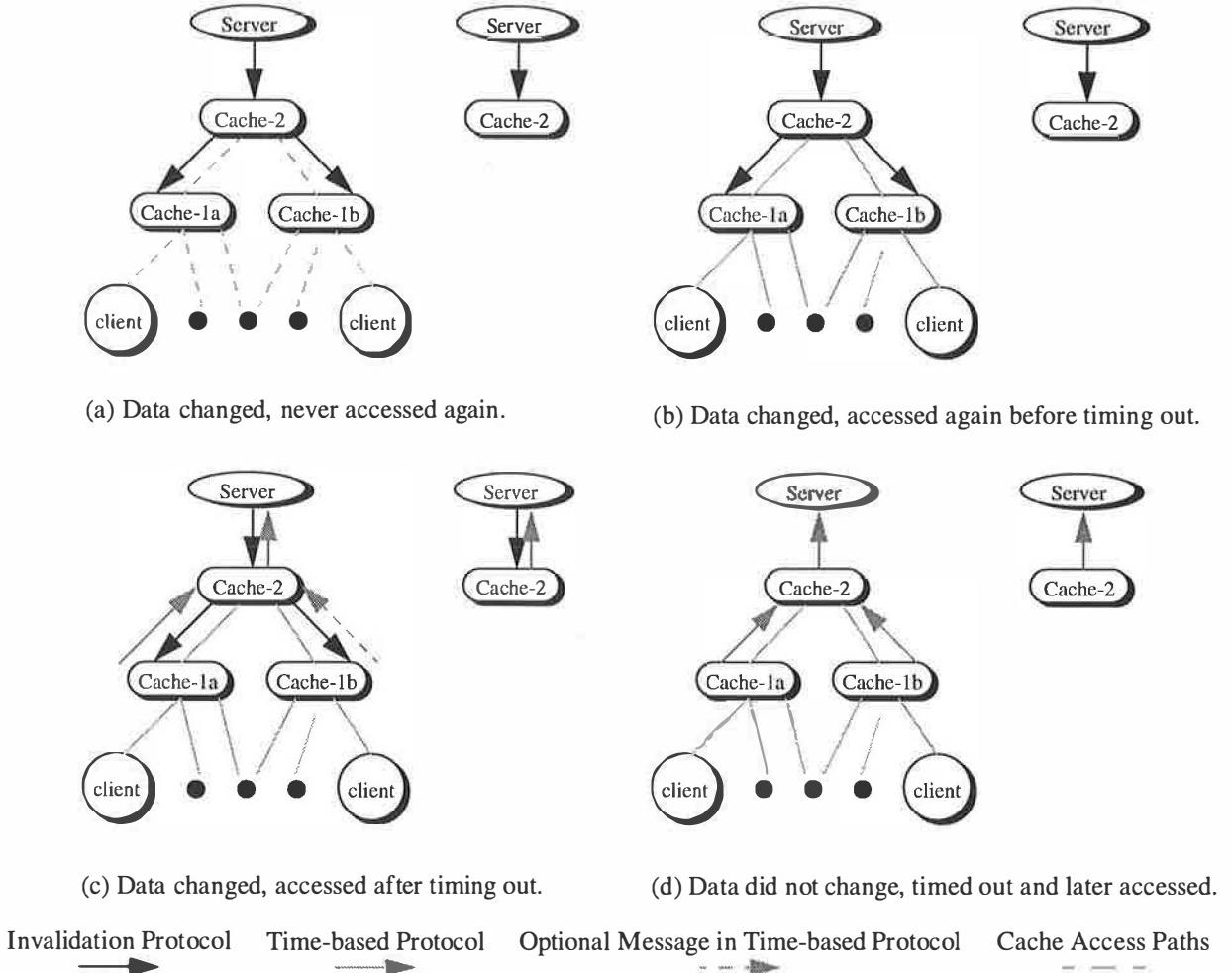


Figure 1. Comparison of Hierarchical Caching and Collapsed Caching. In each diagram, the left picture shows a hierarchical cache and the right picture shows the collapsed hierarchy. The arrows indicate messages sent by the different protocols. In figures a and b, there is no traffic for time-based protocols because the data's time-out has not expired. Therefore, in both simulations, the time-based protocol uses 0% of the bandwidth of the invalidation protocol. In figure c, both time-based and invalidation messages (and files) are sent. If the item is requested from all caches, then the bandwidths of the invalidation protocol and the time-based protocol are equal to each other and the time-based protocol requires 100% of the bandwidth of the invalidation protocol. If some of the caches do not later access the data (e.g. cache-1b), then the time-based protocol will require less bandwidth in the hierarchical model, but still 100% of the bandwidth in the collapsed model. Therefore, when this occurs, we bias the results against the time-based protocols. Figure d shows a similar effect. There is no invalidation traffic, but time-based messages are issued. In the hierarchical case, messages will only be issued from those caches requiring the item, while in the collapsed case, “all” clients request the item. Again, this biases the results *against* the time-based protocols.

possible. We call the simulator with this modification the *optimized simulator*.

Our last change addressed the workload issue mentioned in Section 2. Worrell modeled the file lifetime distribution as a flat distribution between the minimum and maximum observed lifetimes. This means that files were modified with no attention to their type or past modification history. The results of trace analysis from a modified campus Web server show that this is an inappropriate model. Files tend to

exhibit bimodal lifetimes. Either a file will remain unmodified for a long period of time or it will be modified frequently within a short time period [10]. (It was this observation that led us to believe that the Alex protocol would be well suited to Web cache consistency.) Additionally, Worrell used a uniform distribution to generate file requests, but Bestavros has shown that the more popular a file is, the less frequently the file changes [4]. We modified the simulator to use a trace-driven workload. This

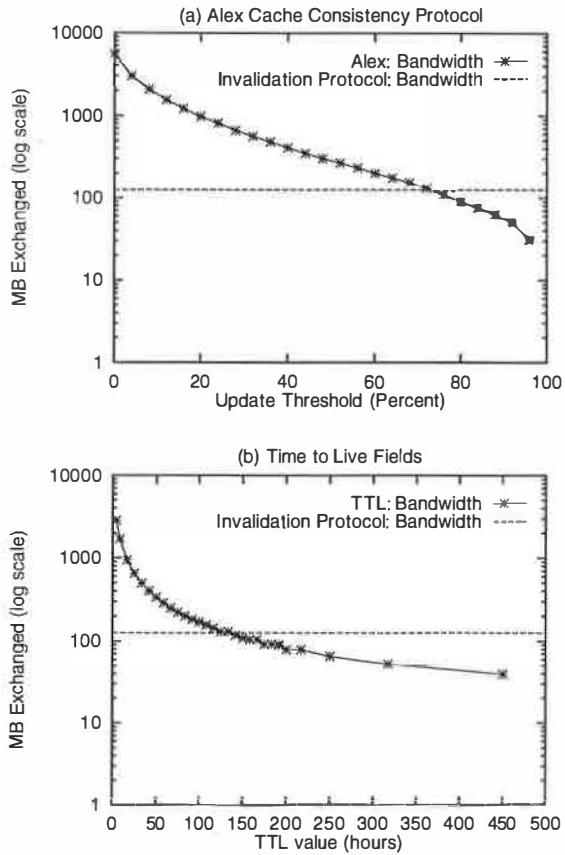


Figure 2. Comparison of bandwidth usage in the base simulator. The cache is pre-loaded with valid copies of all the files held in the primary server. Note the use of a log-scale to display the bandwidth with higher accuracy. The invalidation protocol is superior to both TTL and Alex until the update threshold or TTL is quite large. This result is similar to Worrell's result for TTL protocols and indicates that Alex behaves comparably.

simulator is referred to as the *modified workload simulator*.

4.0 Simulation Results

Figures 2 and 3 show the trade-offs inherent in the parameterization of the Alex and TTL protocols. With Alex, as the update threshold increases, the bandwidth savings also increase (i.e. total bandwidth decreases). However, with this increase in bandwidth savings comes an increase in the number of times stale data is returned to the user (the "stale hits" line in Figure 3). Similarly, with TTL fields, the increase in TTL that induces more bandwidth savings also induces more stale hits. The invalidation protocol is unaffected by parameterization, yielding the constant bandwidth

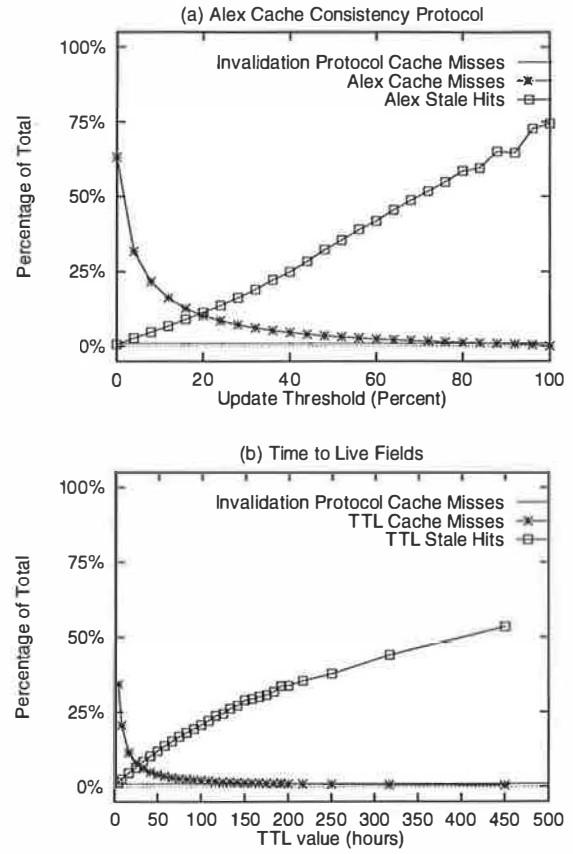


Figure 3. Comparison of cache miss rates in the base simulator. The increases in update threshold and TTL that induced bandwidth savings in Figure 2 also induce an increase in the stale hit rate. The invalidation protocol provides perfect consistency resulting in a 0% stale hit rate (not shown in the figure).

shown in Figure 2, and since valid entries are never evicted from the cache, it also produces the near perfect cache miss rates shown in Figure 3.

Although we expected Alex to outperform TTL, the two figures show that for a specified acceptable stale hit rate, TTL provides greater bandwidth savings. For example, if the acceptable stale hit rate is 25%, then Alex must select an update threshold of approximately 40% (from Figure 3a), inducing a total bandwidth of 400 MB (from Figure 2a). In contrast, to achieve a 25% stale hit rate, the TTL must be set to approximately 125 hours, resulting in a total bandwidth of approximately 130 MB. In both cases, the bandwidth required is greater than that required for the invalidation protocol, and the stale cache rate of 25% is unacceptably high. The difference in bandwidth consumption between Alex and TTL is discussed in more detail in Section 4.2.

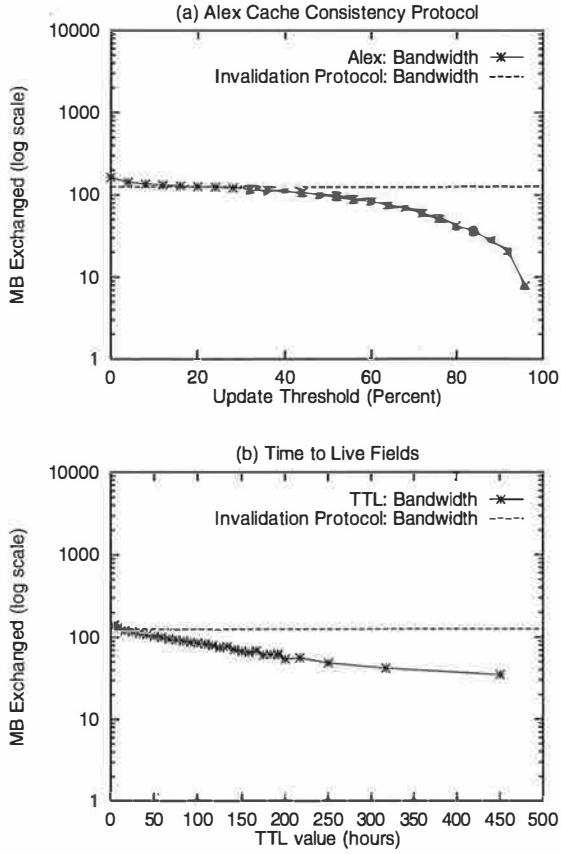


Figure 4. Comparison of bandwidth usage in the optimized simulator. Files are transmitted only when they are truly stale. With this optimization, both TTL and Alex use less bandwidth than the Invalidation Protocol in nearly all cases.

4.1 Optimized Retrieval

Our next set of experiments evaluated the conditional retrieval provided by the optimized simulator. The Alex and TTL protocols query the server to determine the validity of their cached, but invalid, data before requesting that a new copy be sent. Figure 4 shows the effect of this change on the total network bandwidth. With this optimization, both protocols outperform the Invalidation Protocol for most parameter settings.

To understand why the protocols save bandwidth, consider the amount of information that must be exchanged in each case. The information can be categorized into messages and file transfers. The invalidation protocol sends an invalidation message every time that a file changes, but sends files only when an invalid file is requested. Both Alex and TTL send messages only after a file has timed out and has been requested again, and send files only when a file that is truly out of date is requested. All three

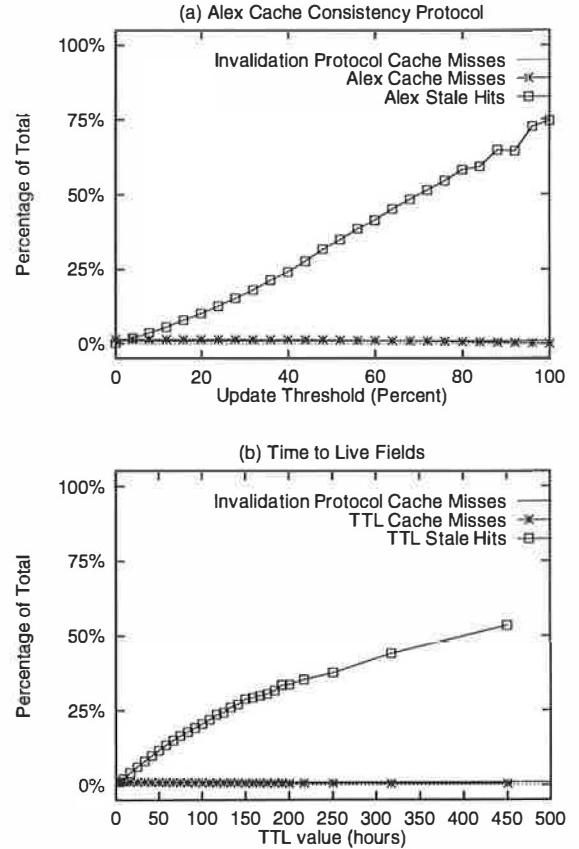


Figure 5. Comparison of cache miss rates in the optimized simulator. The cache miss rates improve dramatically from Figure 3 since invalidated files are left in the cache. All three protocols show miss rates that are indistinguishable from the near perfect miss rate of the invalidation protocol. However, the stale hit rate remains unacceptably high.

protocols transfer files that are truly invalid. Therefore, neither Alex nor TTL will ever transmit more file information than the invalidation protocol, but could transmit less if stale files are ever returned. On the other hand, the amount of bandwidth consumed by Alex and TTL for messages (queries to the server to check for stale data) could be more or less than that for the invalidation protocol depending on the cache settings. Since each message averages 43 bytes and each file averages several thousand bytes, the effect of saving file transfers is much more pronounced than the effect of sending more server queries. As the number of stale hits increases, the bandwidth consumption decreases.

The more dramatic improvement occurs in the miss rates shown in Figure 5. Both Alex and TTL now achieve near perfect miss rates because the invalidated data are left in the cache, avoiding future

Server	Files	Requests	% Remote Requests	Total Changes	% Mutable Files	% Very Mutable Files
DAS	1403	30,093	84%	321	6.83%	2.61%
FAS	290	56,660	39%	11	2.41%	0.00%
HCS	573	32,546	50%	260	23.3%	5.22%

Table 1: Summary of mutability statistics for various campus servers over a one-month period. Mutable files are defined to be those files that were observed to change more than once over the time period. Very mutable files are those that were observed to change more than 5 times. Any request that was not generated by a client in our campus domain was considered remote, and any files added in the middle of this time period were not included in these statistics. Notice that the most popular server, the FAS server, is also the one with the fewest mutable files.

retrievals. Cache misses are recorded only when a file actually needs to be transferred to the cache. Unfortunately, the stale cache hit rate is unchanged. For example, selecting a TTL of 100 hours saves only 32% of the invalidation protocol’s bandwidth but results in a 20% stale cache hit rate. This number of stale hits is probably unacceptable for the moderate bandwidth savings.

4.2 Modified Access Patterns

We expected that an adaptive protocol such as the Alex protocol would do better than the static TTL protocol, so we examined the factors that contributed to Alex’s poor performance. The flat lifetime distribution coupled with the fact that all files were assigned equal retrieval probability seemed to be the leading cause. The analysis of traces gathered in our local environment coupled with results by Bestavros [3] convinced us to consider an alternative workload generator.

Bestavros found that on any given server only a few files change rapidly. Furthermore, he observed that globally popular files are the least likely to change. A workload modeled by these characteristics departs significantly from the workload modeled by the base and optimized simulators. If the file request distribution is skewed towards popular files and popular files change less often, then the number of stale hits reported will decrease significantly. An adaptive protocol, such as Alex, will then work well on both rapidly changing files as well as stable ones. While files are changing rapidly, Alex checks frequently; once the files stabilize, Alex checks infrequently.

The modified workload simulator uses Web server logs from our local environment to generate file lifetimes. The server logs were taken from several campus Web servers, modified to store the *last-modified* timestamps with each file request satisfied

by the servers. We used the file system’s last modification time for the timestamp. The server logs are summarized in Table 1.

The three systems from which we gathered logs are FAS, our university web server, DAS, the web server for the Division of Applies Sciences (think, “College of Engineering”), and HCS, the web server for our local computer society. The statistics from these server logs confirm Bestavros’ observation that the most popular files are also the least mutable ones.

It is instructive to compare our trace characteristics with those of the workload simulated by the base simulator. The traced files change far less often than the files with randomly generated lifetimes. For example, one run of the base simulator included accesses to 2085 files over a 56 day simulated run. Those 2085 files changed 19,898 times yielding a 17% average probability that on any given day a particular file changed. Our HCS trace, which changed the most frequently, involved 573 files changing 260 times over 25 days. This yields a 1.8% average change probability, which is consistent with Bestavros’ per-day file-change probability of 0.5% – 2.0%, with more popular files changing less often than other files.

While the simulation of our trace data modeled the exact modification behavior on our servers, the change probability computed above is based on a small sample size. Bestavros offers another data point, but it is only accurate between one-day intervals. It is possible that the one day granularity masked a number of changes equivalent to those used by Worrell, but it is unlikely, since Bestavros’ data reflected an order of magnitude less change than the simulated workload. Each file that was recorded as changed would have had to have changed not once, but 10 times between samples to produce an equivalent rate of change. Given the significant difference in the rapidity of change between the trace

data and the simulated workload, we expected to observe far fewer stale cache hits with the Alex and TTL protocols using the trace data than we did with the random lifetime generation.

In order to verify that the data from our traces is representative of “typical” web usage, we gathered both information on the distribution of accesses to different types of files as well as the average life-spans of these file types. We gathered this data from two different sources. We obtained information about the distribution of accesses to different types of web objects from a proxy cache at Microsoft. We obtained information about the life-span of different file types from modification logs of the Boston University web server.

The Microsoft proxy cache sits between all Microsoft employees and anything outside of Microsoft. The access logs for the server contain the types and sizes of files accessed, but not the last-modified date for files retrieved, so we could not simulate this log. Instead, we used the data to characterize access patterns by file type. On an average week day, the Microsoft proxy cache server receives approximately 150,000 requests for web objects. Of these, 65% are for image files (gif and jpg). The file type breakdown is shown in the second and third columns of Table 2.

To understand what files are the most likely to change, we analyzed the data gathered from the Boston University web server. Each day between March 28 and October 7, Bestavros sampled the server and recorded all the files that were modified since the previous day [4]. The logs contain approximately 2,500 file references and 14,000 changes during that 186 day time period. Categorizing this data by file type, we can determine the average life span per file type. This data is shown in the last two columns of Table 2.

In computing these life-spans, we err on the side of conservatism, overestimating the rate of change by assuming that all data changed at least once during the measurement interval. This biases the results because the longest life-span we consider is 186 days and there almost certainly exist files with longer life-spans. However, ignoring files that did not change and considering only those files that did change would have skewed the results far more.

Images, which represent 65% of the accesses in the Microsoft data, have the longest lifetimes, living 85–100 days. Surprisingly, image files are also relatively small, so caching them is feasible. This supports our hypothesis that weak consistency caching will be effective, since the most popular web objects also have the longest life-span.

File type	Microsoft		Boston University	
	%-age of total accesses	Average file size	Average life-span (days)	Median Age (days)
gif	55%	7791	85	146
html	22%	4786	50	146
jpg	10%	21608	100	72
cgi	9%	5980	NA	NA
other	4%	NA	NA	NA

Table 2: Tabulation of Microsoft and Boston University server log summaries. The Microsoft data provides information on file access patterns while the Boston University data provides information on file type lifetimes.

While we still need to collect better data from a single server, the behavior observed at Microsoft and Boston University convinced us that our own local traces were representative of the rate of change observed on the web. We then simulated the three different consistency algorithms using a workload based on the trace data summarized in Table 1.

Figures 6 and 7 show dramatically different results from those in Figures 2 through 5. Both Alex and TTL produce less bandwidth usage than the invalidation protocol with few stale cache hits, reflecting the fact that few files change frequently on the server. Since files do not change often, they do not cause stale data to be returned. In contrast to the earlier calculations, we find that with an acceptable stale hit rate of less than 5%, both Alex and TTL demand less bandwidth than the invalidation protocol for nearly all parameter settings and that Alex and TTL offer similar savings in bandwidth.

Having established that the weakly consistent schemes are competitive with invalidation protocols in terms of bandwidth and stale cache hits, it is useful to examine the server load created by the various protocols. Figure 8 shows the number of server operations (i.e. requests for documents, queries to determine whether documents are stale, and invalidation messages) for each of the protocols. While TTL offers bandwidth savings and acceptably low stale cache hit rates, it induces a higher server load than either Alex (with properly tuned parameters) or the invalidation protocol. The number of server queries generated by Alex with an update threshold of 0 is particularly noteworthy. This configuration represents the case where the cache checks with the server on every client request as some poorly designed servers currently do. Not only is this unnecessary, since an update threshold as low as 5%

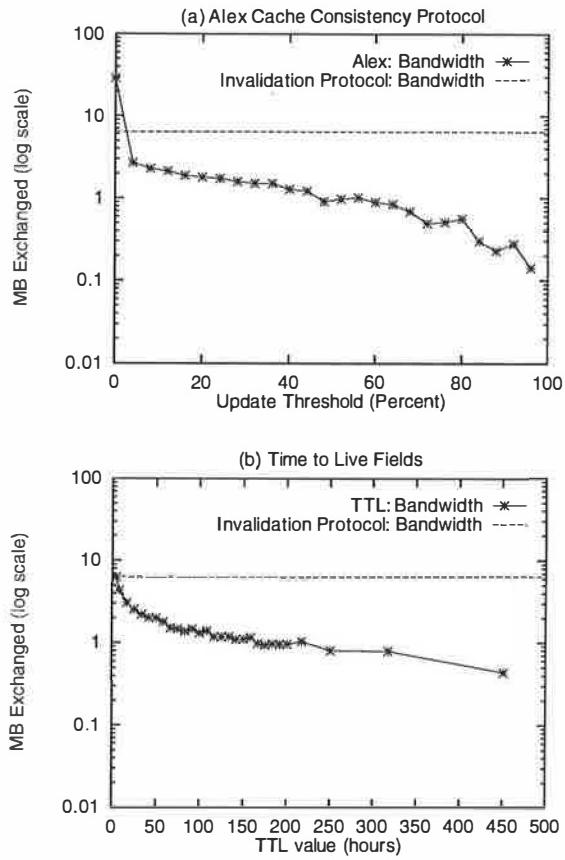


Figure 6. Comparison of bandwidth using the workload modified simulator. These results depict the averages of the FAS, HCS, and DAS traces. Files that were not in the primary host at the beginning of the month were not simulated. Both Alex and TTL use less bandwidth than the Invalidation Protocol for nearly all parameter settings.

returns stale data less than 1% of the time, but it is excessively wasteful of server resources since it creates nearly two orders of magnitude more server queries.

5.0 Future Work

Our simulations indicate that maintaining cache consistency in the World Wide Web need not be expensive. However, there are still important issues to be examined. The time-based protocols (Alex and TTL) both rely on careful tuning of parameters. Leaving this tuning to manual intervention is guaranteed to result in suboptimal performance. Furthermore, as the Boston University and Microsoft trace data indicate, different types of files exhibit different update behavior. One important area of further research is to investigate tuning cache consistency protocol parameters.

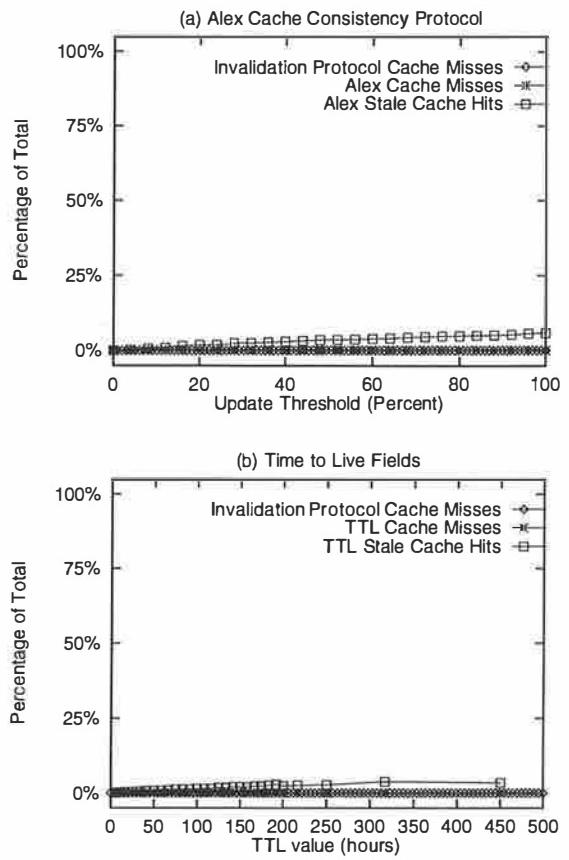


Figure 7. Comparison of cache miss rates using the modified workload simulator. Both protocols provide extremely low stale data rates using trace-driven simulation. The cache miss rates for the invalidation protocol, Alex, and TTL are all less than 0.04%, producing the overlapping lines near 0%.

We are investigating algorithms by which caches can be self-tuning, by adjusting parameters based on the data type and the history of accesses to items of that type.

Another trend in web usage that has an affect on proxy caching is the increasing number of web objects that are dynamically generated. The Microsoft trace logs revealed that 10% of the requests were for dynamically generated pages. This represents a tenfold increase from only six months ago. As the number of dynamic objects increases it will become critical to devise ways to cache the actual scripts that generate dynamic pages. Web scripting languages such as Java and Tcl offer one possible approach, but autonomously replicating the databases that underlie most dynamic content is non-trivial.

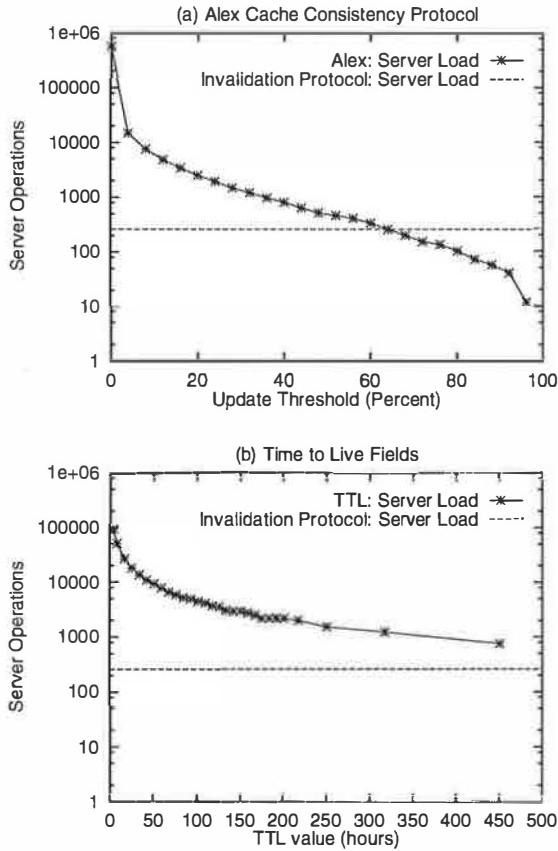


Figure 8. Comparison of server loads on the three consistency protocols. Notice that parameterization is critical for efficient operation of either Alex or TTL and that Alex imposes less load on the server than TTL. TTL always imposes more load than the invalidation protocol while Alex requires an update threshold of at least 64% in order to achieve the same server load as the invalidation protocol. At this 64% threshold, the stale cache miss rate is 4%.

6.0 Conclusions

While Worrell's results presented a strong preference for invalidation protocols relative to TTL, our results differ significantly. If network bandwidth is the driving force, then TTL is an attractive alternative, offering reduced network bandwidth and a low stale hit rate. It does present a significantly higher load to the server, which makes it unattractive. However, in general, the Alex protocol provides the best of all worlds in that it can be tuned to:

- reduce network bandwidth consumption by an order of magnitude over an invalidation protocol,
- produce a stale rate of less than 5%, and

- produce server load comparable to, or less than, that of an invalidation protocol with much less bookkeeping.

Although Alex is preferable to TTL, there are cases where TTL might still be suitable. For example, when object lifetimes are known *a priori*, as is the case with daily news articles or weekly schedules, TTL is the right choice.

Although invalidation protocols are still required when perfect cache consistency is a necessity, the weakly consistent protocols are particularly attractive for a number of reasons. They are both much simpler to implement. They are both more fault resilient when machines become unreachable; the right thing automatically happens. Documents eventually become invalidated and the server is contacted upon subsequent requests. With an invalidation protocol, recovery is much more complicated. The changes required to implement an invalidation protocol in existing web servers and clients is more significant than the effort to implement either TTL or Alex.

7.0 Acknowledgments

We would like to thank Kurt Worrell for inspiring this work and for giving us his simulator, the Harvard Arts and Sciences Computer Services Organization, the Division of Applied Sciences computing staff, and the Harvard Computer Society for running our modified web server, the Microsoft Gibralter team for providing us with trace logs, Keith Smith and Anna Watson for their careful reviews, and Azer Bestavros for his data and continued excellent advice.

8.0 Bibliography

- [1] Andreessen, M., private email correspondence.
- [2] Berners-Lee, T., "Hypertext Transfer Protocol HTTP/1.0," HTTP Working Group Internet Draft, October 14, 1995.
- [3] Bestavros, A., "Demand-based Resource Allocation to Reduce Traffic and Balance Load in Distributed Information Systems," to appear in *Proceedings of the SPDP'95: The 7th IEEE Symposium on Parallel and Distributed Processing*, San Antonio, TX, October 1995.
- [4] Bestavros, A., "Speculative Data Dissemination and Service to Reduce Server Load, Network Traffic and Service Time for Distributed Information Systems," *Proceedings of 1996 International Conference on Data Engineering*, New Orleans, Louisiana, March 1996.

- [5] Blaze, M., "Caching in Large-Scale Distributed File Systems," Princeton University Technical Report, TR-397-92, January 1993.
- [6] Cate, V., "Alex— A Global Filesystem," *Proceedings of the 1992 USENIX File System Workshop*, Ann Arbor, MI, May 1992, 1–12.
- [7] Chankhunthod, A., Danzig, P., Neerdaels, C., Schwartz, M., Worrell, K., "A Hierarchical Internet Object Cache," *Proceedings of the 1996 USENIX Technical Conference*, San Diego, CA, January 1996.
- [8] Dahlin, M., Mather, C., Wang, R., Anderson, T., Patterson, D., "A Quantitative Analysis of Cache Policies for Scalable File Systems," *Proceedings of the 1994 Sigmetrics Conference*, May 1994, 150–160.
- [9] Danzig, P., Hall, R., Schwartz, M., "A Case for Caching File Objects Inside Internetworks," Technical Report, University of Colorado, Boulder, CU-CS-642-93, 1993.
- [10] Gwertzman, J., "Autonomous Replication in Wide-Area Distributed Information Systems," Technical Report TR-95-17, Harvard University Division of Applied Sciences, Center for Research in Computing Technology, 1995.
- [11] Howard, J., Kazar, M., Menees, S., Nichols, D., Satyanarayanan, M., Sidebotham, R., West, M., "Scale and Performance in a Distributed File System," *ACM Transactions on Computer Systems*, 6, 1, February 1988, 51–81.
- [12] Luotonen, A., Frystyk, H., Berners-Lee, T., "W3C httpd," [http://www.w3.org/hypertext/WWW/Daemon/St atus.html](http://www.w3.org/hypertext/WWW/Daemon>Status.html).
- [13] Nelson, M., Welch, B., Ousterhout, J., "Caching in the Sprite Network File System," *ACM Transactions on Computer Systems*, 6, 1, February 1988, 134–154.
- [14] Sandberg, R., Goldberg, D., Kleiman, S., Walsh, D., and Lyon, B., "Design and Implementation of the Sun Network Filesystem," *Proceedings of the Summer 1985 USENIX Conference*, Portland OR, June 1985, 119–130.
- [15] Wessels, D., "Intelligent Caching for World-Wide Web Objects," *Proceedings of INET-95*, 1995.
- [16] Worrell, K., "Invalidation in Large Scale Network Object Caches," Master's Thesis, University of Colorado, Boulder, 1994.

James Gwertzman is a program manager at Microsoft Corporation where he works on the Microsoft Network. His research interests include distributed systems, online communities, and data replication. He received an A.B. degree from Harvard

College in 1995, and was the recipient of a Hoopes prize for his senior thesis. He promises to attend graduate school in the near future.

Margo I. Seltzer is an Assistant Professor of Computer Science in the Division of Applied Sciences at Harvard University. Her research interests include file systems, databases, and transaction processing systems. She is the author of several widely-used software packages including database and transaction libraries and the 4.4BSD log-structured file system. Dr. Seltzer spent several years working at start-up companies designing and implementing file systems and transaction processing software and designing microprocessors. She is a Sloan Foundation Fellow in Computer Science and was the recipient of the University of California Microelectronics Scholarship, The Radcliffe College Elizabeth Cary Agassiz Scholarship, and the John Harvard Scholarship. Dr. Seltzer received an A.B. degree in Applied Mathematics from Harvard/Radcliffe College in 1983 and a Ph. D. in Computer Science from the University of California, Berkeley, in 1992.

A Hierarchical Internet Object Cache

Anawat Chankhunthod

Peter B. Danzig

Chuck Neerdaels

Computer Science Department
University of Southern California

Michael F. Schwartz

Kurt J. Worrell

Computer Science Department
University of Colorado - Boulder

Abstract: This paper discusses the design and performance of a hierarchical proxy-cache designed to make Internet information systems scale better. The design was motivated by our earlier trace-driven simulation study of Internet traffic. We challenge the conventional wisdom that the benefits of hierarchical file caching do not merit the costs, and believe the issue merits reconsideration in the Internet environment.

The cache implementation supports a highly concurrent stream of requests. We present performance measurements that show that our cache outperforms other popular Internet cache implementations by an order of magnitude under concurrent load. These measurements indicate that hierarchy does not measurably increase access latency. Our software can also be configured as a Web-server accelerator; we present data that our *hupd-accelerator* is ten times faster than Netscape's Netsite and NCSA 1.4 servers.

Finally, we relate our experience fitting the cache into the increasingly complex and operational world of Internet information systems, including issues related to security, transparency to cache-unaware clients, and the role of file systems in support of ubiquitous wide-area information systems.

1 Introduction

Perhaps for expedience or because software developers perceive network bandwidth and connectivity as free commodities, Internet information services like FTP, Gopher, and WWW were designed without caching support in their core protocols. The consequence of this misperception now haunts popular WWW and FTP servers. For example, NCSA, the home of Mosaic, moved to a multi-node cluster of servers to meet demand. NASA's Jet Propulsion Laboratory wide-area network links were saturated by the demand for Shoemaker-Levy 9 comet images in July 1994, and Starwave corporation runs a five-node SPARC-center 1000 just to keep up with demand for college basketball scores. Beyond distributing load away from server "hot spots", caching can also save bandwidth, reduce latency, and protect the network from clients that erroneously loop and generate repeated requests [9].

This paper describes the design and performance of the Harvest [5] cache, which we designed to make Internet information services scale better. The cache implementation is

optimized to support a highly concurrent stream of requests with minimal queuing for OS-level resources, using non-blocking I/O, application-level threading and virtual memory management, and a Domain Naming System (DNS) cache. Because of its high performance, the Harvest cache can also be paired with existing HTTP servers (httpd's) to increase document server throughput by an order of magnitude.

Individual caches can be interconnected hierarchically to mirror an internetwork's topology, implementing the design motivated by our earlier NSFNET trace-driven simulation study [10].

1.1 Hierarchical Web versus File System Caches

Our 1993 study of Internet traffic showed that hierarchical caching of FTP files could eliminate half of all file transfers over the Internet's wide-area network links. [10]. In contrast, the hierarchical caching studies of Blaze and Alonso [2] and Muntz and Honeyman [17] showed that hierarchical caches can, at best, achieve 20% hit rates and cut file server workload in half. We believe the different conclusions reached by our study and these two file system studies lay in the workloads traced.

Our study traced wide-area FTP traffic from a switch near the NSFNET backbone. In contrast, Blaze and Alonso [2] and Muntz and Honeyman [17] traced LAN workstation file system traffic. While workstation file systems share a large, relatively static collection of files, such as *gcc*, the Internet exhibits a high degree of read-only sharing among a rapidly evolving set of popular objects. Because LAN utility files rarely change over a five day period, both [17] and [2] studies found little value of hierarchical caching over flat file caches at each workstation: After the first reference to a shared file, the file stayed in the local cache indefinitely and the upper-level caches saw low hit rates.

In contrast to workstation file systems, FTP, WWW, and Gopher facilitate read-only sharing of autonomously owned and rapidly evolving object spaces. Hence, we found that over half of NSFNET FTP traffic is due to sharing of read-only objects [10] and, since Internet topology tends to be organized hierarchically, that hierarchical caching can yield a 50% hit rate and reduce server load dramatically. Claffy and Braun reported similar statistics for WWW traffic [7], which has displaced FTP traffic as the largest contributor to Internet traffic. . .

Second, the cost of a cache miss is much lower for Internet information systems than it is for traditional caching applications. Since a page fault can take 10^5 times longer to service than hitting RAM, the RAM hit rate must be 99.99% to keep the average access speed at twice the cost of a RAM

hit. In contrast, the typical miss-to-hit cost ratio for Internet information systems is 10:1¹, and hence a 50% hit ratio will suffice to keep the average cost at twice the hit cost.

Finally, Internet object caching addresses more than latency reduction. As noted above and in the file system papers, hierarchical caching moves load from server hot spots. Not mentioned in the file system papers, many commercial sites proxy *all* access to the Web and FTP space through proxy caches, out of concern for Internet security. Many Internet sites are forced to use hierarchical object caches.

The Harvest cache has been in use for 1.5 years by a growing collection of about 100 sites across the Internet, as both a proxy-cache and as an httpd-accelerator. Our experiences during this time highlight several important issues. First, cache policy choices are made more difficult because of the prevalence of information systems that provide neither a standard means of setting object Time-To-Live (TTL) values, nor a standard for specifying objects as non-cacheable. For example, it is popular to create WWW pages that modify their content each time they are retrieved, by returning the date or access count. Such objects should not be cached. Second, because it is used in a wide-area network environment (in which link capacity and congestion vary greatly), cache topology is important. Third, because the cache is used in an administratively decentralized environment, security and privacy are important. Fourth, the widespread use of location-dependent names (in the form of Uniform Resource Locators, or URLs) makes it difficult to distinguish duplicated or aliased objects. Finally, the large number of implementations of both clients and servers leads to errors that worsen cache behavior.

We discuss these issues in more depth below.

2 Design

This section describes our design to make the Harvest cache fast, efficient, portable, and transparent.

2.1 Cache Hierarchy

To reduce wide-area network bandwidth demand and to reduce the load on Internet information servers, caches resolve misses through other caches higher in a hierarchy, as illustrated in Figure 1. In addition to the parent-child relationships illustrated in this figure, the cache supports a notion of *siblings*. These are caches at the same level in the hierarchy, provided to distribute cache server load.

Each cache in the hierarchy independently decides whether to fetch the reference from the object's home site or from its parent or sibling caches, using a simple *resolution protocol* that works as follows.

If the URL contains any of a configurable list of substrings, then the object is fetched directly from the object's home, rather than through the cache hierarchy. This feature is used to force the cache to resolve non-cacheable ("cgi-bin") URLs and local URLs directly from the object's home. Similarly, if the URL's domain name matches a configurable list of substrings, then the object is resolved through the particular parent bound to that domain.

Otherwise, when a cache receives a request for a URL that misses, it performs a remote procedure call to all of its siblings and parents, checking if the URL hits any sibling or

¹This rough estimate is based on the observation that it takes about one second for a browser like Netscape to load an object from disk and render it for display, while a remote object takes about 10 seconds to be retrieved and displayed.

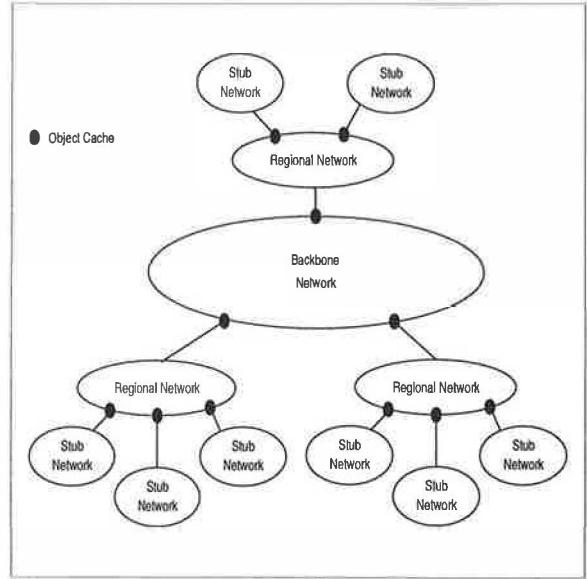


Figure 1: Hierarchical Cache Arrangement.

parent. The cache retrieves the object from the site with the lowest measured latency.

Additionally, a cache option can be enabled that tricks the referenced URL's home site into implementing the resolution protocol. When this option is enabled, the cache sends a "hit" message to the UDP echo port of the object's home machine. When the object's home echos this message, it looks to the cache like a hit, as would be generated by a remote cache that had the object. This option allows the cache to retrieve the object from the home site if it happens to be closer than any of the sibling or parent caches.

A cache resolves a reference through the first sibling, parent, or home site to return a UDP "Hit" packet or through the first parent to return a UDP "Miss" message if all caches miss and the home's UDP "Hit" packet fails to arrive within two seconds. However, the cache will not wait for a home machine to time out; it will begin transmitting as soon as all of the parent and sibling caches have responded. The resolution protocol's goal is for a cache to resolve an object through the source (cache or home) that can provide it most efficiently. This protocol is really a heuristic, as fast response to a ping indicates low latency. We plan to evolve to a metric that combines both response time and available bandwidth.

As will be shown in Section 3.5, hierarchies as deep as three caches add little noticeable access latency. The only case where the cache adds noticeable latency is when one of its parents fail, but the child cache has not yet detected it. In this case, references to this object are delayed by two seconds, the parent-to-child cache timeout. As the hierarchy deepens, the root caches become responsible for more and more clients. To keep root caches servers from becoming overloaded, we recommend that the hierarchy terminate at the first place in the regional or backbone network where bandwidth is plentiful.

2.2 Cache Access Protocols

The cache supports three access protocols: *encapsulating*, *connectionless*, and *proxy-http*. The *encapsulating* protocol encapsulates cache-to-cache data exchanges to permit

end-to-end error detection via checksums and, eventually, digital signatures. This protocol also enables a parent cache to transmit an object's remaining time-to-live to the child cache. The cache uses the UDP-based *connectionless* protocol to implement the parent-child resolution protocol. This protocol also permits caches to exchange small objects without establishing a TCP connection, for efficiency. While the *encapsulating* and *connectionless* protocols both support end-to-end reliability, the *proxy-http* protocol is the protocol supported by most Web browsers. In that arrangement, clients request objects via one of the standard information access protocols (FTP, Gopher, or HTTP) from a cache process. The term "proxy" arose because the mechanism was primarily designed to allow clients to interact with the WWW from behind a firewall gateway.

2.3 Cacheable Objects

The wide variety of Internet information systems leads to a number of cases where objects should not be cached. In the absence of a standard for specifying TTLs in objects themselves, the Harvest cache chooses not to cache a number of types of objects. For example, objects that are password protected are not cached. Rather, the cache acts as an application gateway and discards the retrieved object as soon as it has been delivered. The cache similarly discards URL's whose name implies the object is not cacheable (e.g. <http://www.xyz.com/cgi-bin/query.cgi>, for details about cacheable objects.). It is possible to limit the size of the largest cacheable object, so that a few large FTP objects do not purge ten thousand smaller objects from the cache.

2.4 Unique Object Naming

A URL does *not* name an object uniquely; the URL *plus* the MIME² header issued with the request uniquely identify an object. For example, a WWW server may return a text version of a postscript object if the client's browser is not able to view postscript. We believe that this capability is not used widely, and currently the cache does not insist that the request MIME headers match when a request hits the cache.

2.5 Negative Caching

To reduce the costs of repeated failures (e.g., from erroneously looping clients), we implemented two forms of negative caching. First, when a DNS lookup failure occurs, we cache the negative result for five minutes (chosen because transient Internet conditions are typically resolved this quickly). Second, when an object retrieval failure occurs, we cache the negative result for a parameterized period of time, with a default of five minutes.

2.6 Cache-Awareness

When we started designing the cache, we anticipated *cache-aware* clients that would decide between resolving an object indirectly through a parent cache or directly from the object's home. Towards this end, we created a version of Mosaic that could resolve objects through multiple caches, as illustrated in Figure 2. Within a few months, we reconsidered and dropped this idea as the number of new Web clients blossomed (cello, lynx, netscape, tkwww, etc.).

While no Web client is completely cache-aware, most support access through IP firewalls. Clients send all their

²MIME stands for "Multipurpose Internet Mail Extensions". It was originally developed for multimedia mail systems [4], but was later adopted by HTTP for passing typing and other meta data between clients and servers.

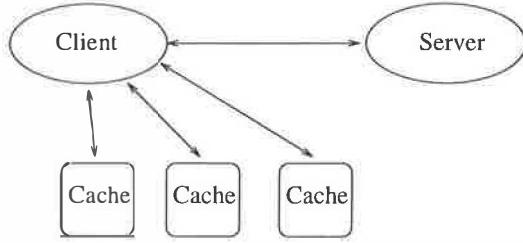


Figure 2: Cache-aware client

requests to their *proxy-server*, and the proxy-server decides how best to resolve it.

There are advantages and disadvantages to the cache-aware and cache-unaware approaches. Cache-unaware clients are simpler to configure; just set the proxy bindings that users already understand and which are needed to provide Web service through firewalls. On the other hand, cache-aware clients would permit load balancing, avoid the single point of failure caused by proxy caching, and (as noted in Section 2.2) allow a wider range of security and end-to-end error checking.

2.7 Security, Privacy, and Proxy-Caching

What is the effect of proxy-caching on Web security and privacy? WWW browsers support various authorization mechanisms, all encoded in MIME headers exchanged between browser and server. The *basic* authorization mechanism involves clear-text exchange of passwords. For protection from eavesdropping, the *Public Key* authorization mechanism is available. Here, the server announces its own public key in clear-text, but the rest of the exchange is encrypted for privacy. This mechanism is vulnerable to IP-spoofing, where a phony server can masquerade as the desired server, but the mechanism is otherwise invulnerable to eavesdroppers. Thirdly, for those who want both privacy and authentication, a *PGP* based mechanism is available, where public key exchange is done externally.

A *basic* authentication exchange follows the following dialog:

1. Client: GET <URL>
2. Server: HTTP:1.0 401 Unauthorized --
Authentication failed
3. Client: GET <URL> Authorization:
<7-bit-encoded name:password>
4. Server: <returns a, b, c or d>
 - a. Reply
 - b. Unauthorized 401
 - c. Forbidden 403
 - d. Not Found 404

Given the above introduction to HTTP security mechanisms, we now explain how the cache transparently passes this protocol between browser and server.

When a server passes a 401 Unauthorized message to a cache, the cache forwards it back to the client and purges the URL from the cache. The client browser, using the desired security model, prompts for a username and password, and reissues the GET URL with the authentication and authorization encoded in the request MIME header. The cache detects the authorization-related MIME header, treats it as

any other kind of non-cacheable object, returns the retrieved document to the client, but otherwise purges all records of the object. Note that under the clear-text *basic* authorization model, anyone, including the cache, could snoop the authorization data. Hence, the cache does not weaken this already weak model. Under the *Public Key* or *PGP* based models, neither the cache nor other eavesdroppers can interpret the authentication data.

Proxy-caching defeats IP address-based authentication, since the requests appear to come from the cache's IP address rather than the client's. However, since IP addresses can be spoofed, we consider this liability an asset of sorts. Proxy-caching does not prevent servers from encrypting or applying digital signature to their documents.

As a final issue, unless Web objects are digitally signed, an unscrupulous system administrator could insert invalid data into his proxy-cache. You have to trust the people who run your caches, just as you must trust the people who run your DNS servers, packet switches, and route servers. Hence, proxy-caching does not seriously weaken Web privacy.

2.8 Threading

For efficiency and portability across UNIX-like platforms, the cache implements its own non-blocking disk and network I/O abstractions directly atop a BSD *select* loop. The cache avoids forking except for misses to FTP URLs; we retrieve FTP URLs via an external process because the complexity of the protocol makes it difficult to fit into our select loop state machine. The cache implements its own DNS cache and, when the DNS cache misses, performs non-blocking DNS lookups (although without currently respecting DNS TTLs). As referenced bytes pour into the cache, these bytes are simultaneously forwarded to all sites that referenced the same object and are written to disk, using non-blocking I/O. The only way the cache will stall is if it takes a virtual memory page fault—and the cache avoids page faults by managing the size of its VM image (see Section 2.9). The cache employs non-preemptive, run-to-completion scheduling internally, so it has no need for file or data structure locking. However, to its clients, it appears multi-threaded.

2.9 Memory Management

The cache keeps all meta-data for cached objects (URL, TTL, reference counts, disk file reference, and various flags) in virtual memory. This consumes 48 bytes + `strlen(URL)` per object on machines with 32-bit words³. The cache will also keep exceptionally hot objects loaded in virtual memory, if this option is enabled. However, when the quantity of VM dedicated to hot object storage exceeds a parameterized high water mark, the cache discards hot objects by LRU until VM usage hits the low water mark. Note that these objects still reside on disk; just their VM image is reclaimed. The hot-object VM cache is particularly useful when the cache is deployed as an *httpd-accelerator* (discussed in Section 3.1).

The cache is write-through rather than write-back. Even objects in the hot-object VM cache appear on disk. We considered memory-mapping the files that represent objects, but could not apply this technique because it would lead to page-faults. Instead, objects are brought into cache via non-blocking I/O, despite the extra copies.

³We plan to replace the variable length URL with a fixed length MD5, reducing this number to 44+16=60 bytes/object.

Objects in the cache are referenced via a hash table keyed by URL. Cacheable objects remain cached until their cache-assigned TTL expires and they are evicted by the cache replacement policy, or the user manually evicts them by clicking the browser's "reload" button (the mechanism for which is discussed in Section 5.1). If a reference touches an expired Web object, the cache refreshes the object's TTL with an HTTP "get-if-modified".

The cache keeps the URL and per-object data structures in virtual memory but stores the object itself on disk. We made this decision on the grounds that memory should buy performance in a server-bottlenecked system: the meta-data for 1,000,000 objects will consume 60-80MB of real memory. If a site cannot afford the memory, then it should use a cache optimized for memory space rather than performance.

2.10 Disk Management

When disk space exceeds the high water mark, the cache enters its garbage collection mode. In this mode, every few dozen cache references, the cache discards the oldest objects encountered in a row of its object hash table. When disk usage drops below the low water mark, the cache exits from its garbage collection mode. If disk usage ever reaches the configured maximum, it immediately discards the oldest objects from the next row of the hash table.

The cache manages multiple disks and attempts to balance load across them. It creates 100 numbered directories on each disk and rotates object creation among the various disks and directories. Hence, an average directory of a cache that manages four disks and a million objects, stores 2500 numbered files. Since directory entries average about 24 bytes, an average directory may grow to 15, 4KB disk blocks. The number of files per directory can be increased to decrease directory size and reduce file system overhead.

3 Performance

We now compare the performance of the Harvest cache against the CERN proxy-HTTP cache [15] and evaluate the Harvest cache's *httpd-accelerator*'s performance gain over the Netscape Netsite, NCSA 1.4, and CERN 3.0 Web servers. We also attempt to attribute the performance improvement to our various design decisions, as laid out in Section 2, and to evaluate the latency degradation of faulting objects through hierarchical caches. The measurements presented in this section were taken on SPARC 20 model 61 and SPARC 10 model 30 workstations. We will see that Harvest's high performance is mostly due to our disk directory structure, our choice to place meta-data in VM, and our threaded design.

3.1 Harvest vs. CERN Cache

To make our evaluation less dependent on a particular hit rate, we evaluate cache performance separately on hits and on misses for a given list of URLs. Sites that know their hit rate can use these measurements to evaluate the gain themselves. Alternatively, the reader can compute expected savings based on hit rates provided by earlier wide-area network traffic measurement studies [10, 7].

Figures 3 and 4 show the cumulative distribution of response times for hits and misses respectively. Figure 3 also reports both the median and average response times. Note that CERN's response time tail extends out to several seconds, so its average is three times its median. In the discussion below, we give CERN the benefit of the doubt and discuss median rather than average response times.

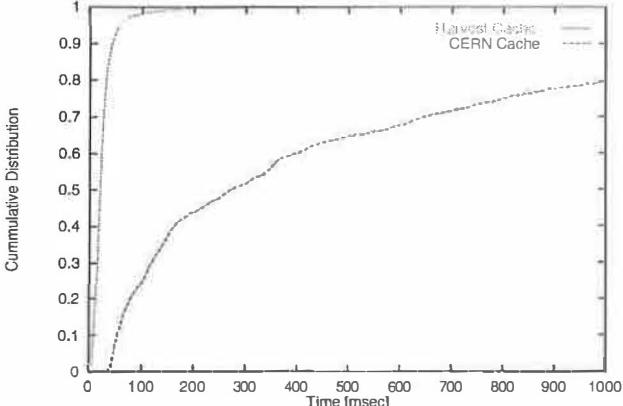


Figure 3: Cumulative distribution of cache response times for hits, ten concurrent clients. The vertical-axis plots the fraction of events that take less than the time recorded on the horizontal-axis. For example, 60% of the time a CERN cache returns a hit in under 500 ms while 95% of the time a Harvest cache returns an object that hits in under 100 ms. CERN's long response time tail makes its average response time significantly larger than its median response time.

To compute Figure 4, ten clients concurrently referenced 200 unique objects of various sizes, types, and Internet locations against an initially empty cache. A total of 2,000 objects were faulted into the cache this way. Once the cache was warm, all ten clients concurrently referenced all 2,000 objects, in random order, to compute Figure 3. No cache hierarchy was used.

These figures show that the Harvest cache is an order of magnitude faster than the CERN cache on hits and on average about twice as fast on misses. We discuss the reasons for this performance difference in Section 3.4. We chose ten concurrent clients to represent a heavily accessed Internet server. For example, the JPL server holding the Shoemaker-Levy 9 comet images received a peak of approximately 4 requests per second, and the objects retrieved ranged from 50-150 kbytes.

For misses there is less difference between the Harvest and CERN caches because response time is dominated by remote retrieval costs. However, note the bump at the upper right corner of Figure 4. This bump comes about because approximately 3% of the objects we attempted to retrieve timed out (causing a response time of 75 seconds)—either due to unreachable remote DNS servers or unreachable remote object servers. While both the Harvest and the CERN caches will experience this long timeout the first time an object retrieval is requested, the Harvest cache's negative DNS and object caching mechanisms will avoid repeated timeouts issued within 5 minutes of the failed request. This can be important for caches high up in a hierarchy because long timeouts will tie up file descriptors and other limited system resources needed to serve the many concurrent clients.

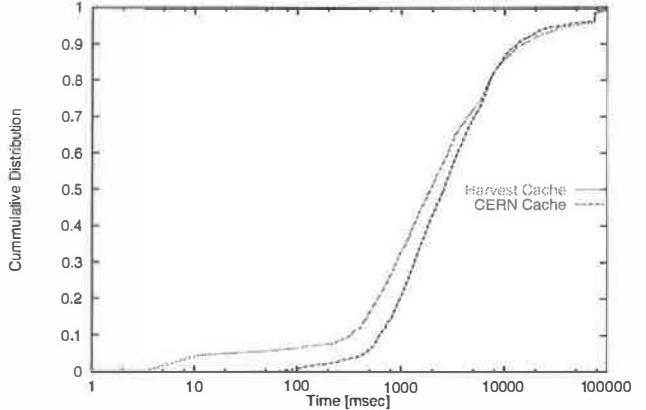


Figure 4: Cumulative distribution of cache response times for misses, ten concurrent clients, 2,000 URLs.

3.2 Httpd-Accelerator

This order of magnitude performance improvement on hits suggests that the Harvest cache can serve as an httpd *accelerator*. In this configuration the cache pretends to be a site's primary httpd server (on port 80), forwarding references that miss to the site's real httpd (on port 81). Further, the Web administrator renames all non-cacheable URLs to the httpd's port (81). References to cacheable objects, such as HTML pages and GIFs are served by the Harvest cache and references to non-cacheable objects, such as queries and cgi-bin programs, are served by the true httpd on port 81. If a site's workload is biased towards cacheable objects, this configuration can dramatically reduce the site's Web workload.

This approach makes sense for several reasons. First, by running the httpd-accelerator in front of the httpd, once an object is in the accelerator's cache all future hits will go to that cache, and misses will go to the native httpd. Second, while the httpd servers process forms and queries, the accelerator makes the simpler, common case fast [14]. Finally, while it may not be practical or feasible to change the myriad httpd implementations to use the more efficient techniques advocated in this paper, sites can easily deploy the accelerator along with their existing httpd.

While the benefit of running an httpd-accelerator depends on a site's specific workload of cacheable and non-cacheable objects, note that the httpd-accelerator cannot degrade a site's performance. Further note that objects that don't appear cacheable at first glance, can be cached at some slight loss of transparency. For example, given a demanding workload, accelerating access to "non-cacheable" objects like sport scores and stock quotes is possible if users can tolerate a short refresh delay, such as thirty seconds.

3.3 Httpd-Accelerator vs. Netsite & NCSA 1.4

Figure 5 demonstrates the performance of a Harvest cache configured as an httpd-accelerator. In this experiment, we faulted several thousand objects into a Harvest cache and then measured the response time of the Harvest cache versus the NCSA and Netsite httpd. Notice how Harvest serves documents that hit the httpd-accelerator with a median of 20 milliseconds, while the medians of Netscape's Netsite and NCSA's 1.4 httpd are each about 300 ms. On objects that miss, Harvest adds only about 20 ms to NCSA's and Netscape's 300 ms median access times.

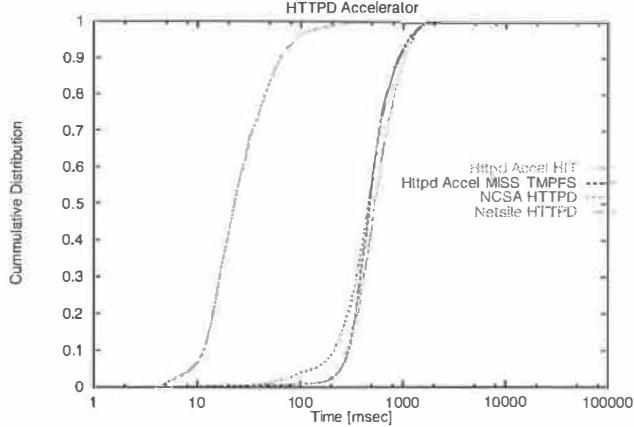


Figure 5: Response time of an httpd-accelerator versus a native httpd for a workload of all hits.

Note that on restart, the httpd-accelerator simply refaults objects from the companion Web server. For this reason, for added performance, the accelerator's disk store can be a memory based filesystem (TMPFS).

The httpd-accelerator can serve 200, small objects per second. We measured this by having 10 clients concurrently fetching two hundred different URL's, in random order, from a warm cache. Note that this is faster than implied by the average response time; we explain this below.

3.4 Decomposing Cache Performance

We now decompose how the Harvest cache's various design elements contribute to its performance. Our goal is to explain the roughly 260 ms difference in median and roughly 800 ms difference in average response times for the "all hits" experiment summarized in Figure 3.

The factor of three difference between CERN's median and average response time, apparent in CERN's long response time tail, occurs because under concurrent access, the CERN cache is operating right at the knee of its performance curve. Much of the response time above the median value corresponds to queueing delay for OS resources (e.g., disk accesses and CPU cycles). Hence, below, we explain the 260 ms difference between CERN's and Harvest's median response times (see Table 1).

Establishing and later tearing down the TCP connection between client and cache contributes a large part of the Harvest cache response time. Recall that TCP's three-way handshakes add a round trip transmission time to the beginning of a connection and a round trip time to the end. Since the Harvest cache can serve 200 small objects per second (5 ms per object) but the median response time as measured by cache clients is 20 ms, this means that 15 ms of the round-trip time is attributable to TCP connection management. This 15 ms is shared by both CERN and the Harvest cache.

We measured the savings of implementing our own threading by measuring the cost to `fork()` a UNIX process that opens a single file (`/bin/ls .`). We measured the savings from caching DNS lookups as the time to perform `gethostbyname()`. DNS lookups of names pre-faulted into a DNS server on the local network. We computed the savings of keeping object meta-data in VM by counting the file system accesses of the CERN cache for retrieving meta-data from the UNIX file system. We computed the savings

Factor	Savings [msec.]
RAM Meta Data	112
Hot Object RAM Cache	112
Threading	36
DNS Lookup Cache	3
Total	264

Table 1: Back of the envelope breakdown of Performance Improvements

from caching hot objects in VM by measuring the file system accesses of the CERN cache to retrieve hot objects, excluding hits from the OS buffer pool.

We first measured the number of file-system operations by driving cold-caches with a workload of 2,000 different objects. We then measured the number of file-system operations needed to retrieve these same 2,000 objects from the warm caches. The first, all-miss, workload measures the costs of writing objects through to disk; the all-hit workload measures the costs of accessing meta-data and objects. Because SunOS instruments NFS mounted file systems better than it instruments file systems directly mounted on a disk, we ran this experiment on an NFS-mounted file system. We found that the CERN cache averages 15 more file system operations per object for meta-data manipulations and 15 more file system operations per object for reading object data. Of course, we cannot convert operation counts to elapsed times because they depend on the size, state and write-back policy of the OS buffer pool and in-core inode table. (In particular, one can reduce actual disk I/O's by dedicating extra memory to file system buffering.) As a grotesquely coarse estimate, Table 1 assumes that disk operations average 15 ms and that half of the file system operations result in disk operations or 7.5 ms average cost per file system operation.

3.5 Cache Hierarchy vs. Latency

The benefits of hierarchical caching (namely, reduced network bandwidth consumption, reduced access latency, and improved resiliency) come at a price. Caches higher in the hierarchy must field the misses of their descendants. If the equilibrium hit rate of a leaf cache is 50%, this means that half of all leaf references get resolved through a second level cache rather than directly from the object's source. If the reference hits the higher level cache, so much the better, as long as the second and third level caches do not become a performance bottleneck. If the higher level caches become overloaded, then they could actually increase access latency, rather than reduce it.

Running on a dedicated SPARC 20, the Harvest cache can respond to over 250 UDP hit or miss queries per second, deliver as many as 200 small objects per second, and deliver 4 Mbits per second to clients. At today's regional network speeds of 1 Mbit/Second, the harvest cache can feed data to users four times faster than the regional network can get the data to the cache. Clearly, a Harvest cache is not a performance bottleneck. As an alternative way to look at the problem, in October 1995 the America Online network served 400,000 objects an hour during peak load. Depending on hit rate, a half dozen Harvest caches can support the entire AOL workload.

Figure 6 shows the response time distribution of faulting an object through zero, one and two levels of hierarchical caching. Figure 6 is read in two parts: access times from

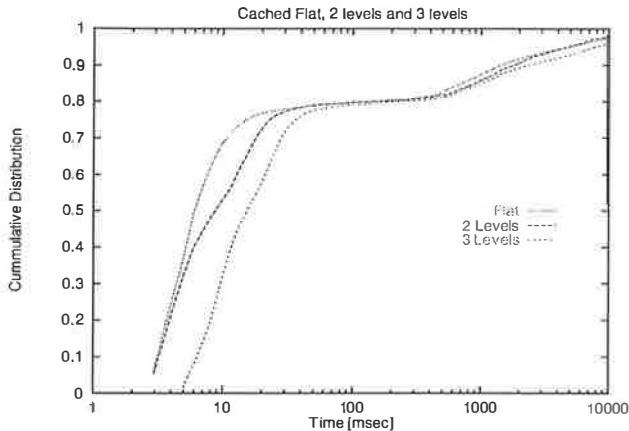


Figure 6: Effect of cache hierarchy on cache response time.

1-20 ms correspond to hits at a first level cache. Access times above 20 ms are due to hierarchical references that wind their way through multiple caches and to the remote Web server. These measurements were gathered using five concurrent clients, each referencing the same 2,000 objects in a random order, against initially cold caches. The caches communicated across an Ethernet LAN, but the references were to distant objects. The result of this workload is that at least one client faults the object through multiple caches, but most of the clients see a first-level hit for that object.

Under this 80% hit workload, the average latency increases a few milliseconds per cache level (pick any point on the CDF axis and read horizontally across the graph until you cross the 1-level, 2-level, and 3-level lines.). For a 0% hit workload, each level adds 4-10 milliseconds of latency. Of course, if the upper-level caches are saturated or if the network connection to the upper-level cache is slow, these latencies will increase.

While this particular experiment does not correspond to any real workload, our point is that cache hierarchies do not significantly reduce cache performance on cache misses.

4 Cache Consistency

The Harvest cache, patterned after the Internet's Domain Naming System, employs TTL-based cache consistency. Like the Domain Naming system and the Alex File System, the Harvest cache can return stale data. If the object's MIME header conveys an expiration time, Harvest respects it. If the object's MIME header conveys its last modification time, Harvest sets the object's TTL to one half the time elapsed since the object was last modified. If the object carries neither time stamp, the cache assigns a default TTL according to the first regular expression that the URL matches. Unlike HTTP which can specify expiration times, Gopher and FTP provide no mechanism for owners to specify TTLs. Hence, when the Harvest cache fetches a Gopher or FTP object, it assigns it a default TTL based on regular expression. When the TTL expires, the cache marks the object as expired. For references to HTTP objects, the cache can revalidate expired objects using a "get-if-modified" fetch. Get-if-modified fetches are propagated correctly through the hierarchy.

We believe that Web caches should not implement hierarchical invalidation schemes, but that sites exporting time-critical data should simply set appropriate expiration times. We say this because hierarchical invalidation, like AFS's

call-back invalidations [13], requires state, and the Internet's basic unreliability and scale complicate stateful services. Below, we present data from two additional studies that support this position.

Object Lifetimes

In the absence of server-specified object expiration times, how should we set the cache's default TTL and how frequently should we "get-if-modified" versus simply returning the cached value? This question has several answers. Some service providers are contractually obligated to keep their cached data consistent: "We promise to get-if-modified if the cached value is over 15 minutes old". Others leave this question to the user and the browser's "reload" command. A quick study of Web data reveals that Internet object lifetimes vary widely and that no single mechanism will suffice.

To illustrate this, we periodically sampled the modification times of 4,600 HTTP objects distributed across 2,000 Internet sites during a three month period. We found that the mean lifetime of all objects was 44 days, HTML text objects and images were 75 and 107 days respectively, and objects of unknown types averaged 27 days. Over 28% of the objects were updated at least every 10 days, and 1% of the objects were dynamically updated. While WWW objects may become less volatile over time, the lifetime variance suggests that a single TTL applied to all objects will not be optimal. Note also that the time between modifications to particular objects is quite variable. This reinforces the need for picking cache TTLs based on last modification times, but that do not grow too large.

In the absence of object owner-provided TTLs, caches could grow object TTLs from some default minimum to some default maximum via binary exponential backoff. Each time an object's TTL expires, it could be revalidated with yet another get-if-modified.

Hierarchical Invalidations

In large distributed environments such as the Internet, systems designers have typically been willing to live with some degree of cache inconsistency to reduce server hot spots, network traffic, and retrieval latency. In reference [19], we evaluate hierarchical invalidation schemes for Internet caching, based on traces of Internet Web traffic and based on the topology of the current U.S. Internet backbone. In the absence of explicit expiration times and last-modification times, this study found that with a default cache TTL of 5 days, 20% of references are stale. Further, it found that the network cost of hierarchical invalidation exceeded the savings of Web caching for default cache TTLs shorter than five days.

Note that achieving a well-working hierarchical invalidation scheme will not be easy. First, hierarchical invalidation requires support from all data providers and caches. Second, invalidation of widely shared objects will cause bursts of synchronization traffic. Finally, hierarchical invalidation cannot prevent stale references, and would require considerable complexity to deal with machine failures.

At present we do not believe hierarchical invalidation can practically replace TTL based consistency in a wide-area distributed environment. However, part of our reluctance to recommend hierarchical invalidation stems from the current informal nature of Internet information. While most data available on the Internet today cause no problems even if stale copies are retrieved, as the Internet evolves to support more mission critical needs, it may make sense to try to

overcome the hurdles of implementing a hybrid hierarchical invalidation mechanism for the applications that demand data coherence.

5 Experience

5.1 Transparency

Of our goals for speed, efficiency, portability and transparency, true transparency was the most difficult to achieve. Web clients expect caches and firewall gateways to translate FTP and Gopher documents into HTML and transfer them to the cache via HTTP, rather than simply forwarding referenced objects. This causes several problems. First, in HTTP transfers, a MIME header specifying an object's size should appear before the object. However, most FTP and Gopher servers cannot tell an object's size without actually transferring the object. This raises the following problem: should the cache read the entire object before it begins forwarding data so that it can get the MIME header right, or should it start forwarding data as soon as possible, possibly dropping the size-specifying MIME header? If the cache reads the entire object before forwarding it, then the cache may inject latency in the retrieval or, worse yet, the client may time out, terminate the transfer and lead the user to believe that the URL is unavailable. We decided not to support the size specification to avoid the timeout problem.

A related problem arises when an object exceeds the configured maximum cacheable object size. On fetching an object, once it exceeds the maximum object size, the cache releases the object's disk store but continues to forward the object to the waiting clients. This feature has the fortunate consequence of protecting the cache from Web applications that send endless streams only terminated by explicit user actions.

Web clients, when requesting a URL, transmit a MIME header that details the viewer's capabilities. These MIME headers differ between Mosaic and Netscape as well as from user to user. Variable MIME headers impact performance and transparency. As it happens, the Mosaic MIME headers range from a few hundred bytes to over a kilobyte and are frequently fragmented into two or more IP packets. Netscape MIME headers are much shorter and often fit in a single IP packet. These seemingly inconsequential details have a major impact that force us to trade transparency for performance.

First, if a user references an object first with Netscape and then re-references it with Mosaic, the MIME headers differ and officially, the cache should treat these as separate objects. Likewise, it is likely that two Mosaic users will, when naming the same URL, generate different MIME headers. This also means that even if the URL is a hit in a parent or sibling cache, correctness dictates that the requested MIME headers be compared. Essentially, correctness dictates that the cache hit rate be zero because any difference in any optional field of the MIME header (such as the user-agent) means that the cached objects are different because a URL does not name an object; rather, a URL plus its MIME header does. Hence, for correctness, the cache must save the URL, the object, *and* the MIME header. Testing complete MIME headers makes the parent-sibling UDP ping protocol expensive and almost wasteful. For these reasons, we do not compare MIME headers.

Second, some HTTP servers do not completely implement the HTTP protocol and close their connection to the client before reading the client's entire MIME header. Their

underlying operating system issues a TCP-Reset control message that leads the cache to believe that the request failed. The longer the client's MIME header, the higher the probability that this occurs. This means that Mosaic MIME headers cause this problem more frequently than Netscape MIME headers. Perhaps for this reason, when it receives a TCP-Reset, Mosaic transparently re-issues the request with a short, Netscape-length MIME header. This leaves us with an unmaskable transparency failure since the cache cannot propagate TCP-Resets to its clients. Instead, the cache returns a warning message that the requested object may be truncated, due to a "non-conforming" HTTP server.

Third, current HTTP servers do not mark objects with a TTL, which would assist cache consistency. With the absence of help from the HTTP servers, the cache applies a set of rules to determine if the requested URL is likely a dynamically evaluated (and hence uncachable) object. Some news services replace their objects many times in a single day, but their object's URLs do not imply that the object is not cacheable. When the user hits the client's "reload" button on Mosaic and Netscape, the client issues a request for the URL and adds a "don't-return-from-cache" MIME header that forces the cache to (hierarchically) fault in a fresh copy of an item. The use of the "reload" button is the least transparent aspect of the cache to users.

Fourth, both Mosaic and Netscape contain a small mistake in their proxy-Gopher implementations. For several months, we periodically re-reported the bug to Netscape Communications Corp., NCSA, and Spyglass, Inc., but none of these organizations chose to fix the bug. Eventually we modified the cache to avoid the client's bugs, forcing the cache to translate the Gopher and FTP protocols into properly formatted HTML.

Note that the Harvest cache's encapsulating protocol (see Section 2.2) supports some of the features that the proxy-*http* protocol sacrifices in the name of transparency. In the future, we may change cache-to-cache exchanges to use the encapsulating protocol.

5.2 Deployment

As Harvest caches get placed in production networks, we continue to learn subtle lessons. Our first such lesson was the interaction of DNS and our negative cache. When users reference distant objects, occasionally the cache's DNS resolver times out, reports a lookup failure, and the cache adds the IP address to its negative IP cache. However, a few seconds later, when the user queries DNS directly, the server returns a quick result. What happened, of course, is that the nameserver resolves the name between the time that the cache gives up and the time that the frustrated user queries the name server directly. Our negative cache would continue to report that the URL was unresolvable for the configured negative caching interval. Needless to say, we quickly and dramatically decreased the default negative cache interval.

At other times Harvest caches a DNS name that it managed to resolve through an inconsistent, secondary name server for some domain. When the frustrated user resolves the name directly, he may get his reply from a different, consistent secondary server. Again the user reports a bug that Harvest declares that a name is bad when it is not.

Caches serving intense workloads overflowed the operating system's open file table or exhausted the available TCP port number space. To solve these problems, we stopped accepting connections to new clients once the we approach the file descriptor or open file table limit. We also added watch-

dog timers to shut down clients or servers that refused to close their connections. As people port the cache to various flavors of UNIX, some had to struggle to get non-blocking disk I/O correctly specified.

Deploying any Web cache—Harvest, Netscape, or CERN—for a regional network or Internet Service Provider is tricky business. As Web providers customize their pages for particular browsers, maintaining a high cache hit rate will become harder and harder.

5.3 Open Systems vs. File Systems

The problems we faced in implementing the Harvest object cache were solved a decade ago in the operating systems community, in the realm of distributed file systems. So the question naturally arises, “Why not just use a file system and dump all of this Web silliness?” For example, Transarc proposes AFS as a replacement for HTTP [18].

AFS clearly provides better caching, replication, management, and security properties than the current Web does. Yet, it never reached the point of exponential growth that characterizes parts of the Internet infrastructure, as has been the case with TCP/IP, DNS, FTP, Gopher, WWW, and many other protocols and services. Why would the Internet community prefer to rediscover and re-implement all of the technologies that the operating systems community long ago solved?

Part of the answer is certainly that engineers like to reinvent the wheel, and that they are naturally lazy and build the simplest possible system to satisfy their immediate goals. But deeper than that, we believe the answer is that the Internet protocols and services that become widespread have two characterizing qualities: simplicity of installation/use, and openness. As a complex, proprietary piece of software, AFS fails both tests.

But we see a more basic, structural issue: We believe that file systems are the wrong abstraction for ubiquitous information systems. They bundle together a collection of features (consistency, caching, etc.) in a way that is overkill for some applications, and the only way to modify the feature set is either to change the code in the operating system, or to provide mechanisms that allow applications selective control over the features that are offered (e.g., using `ioctl`s and kernel build-time options). The Internet community has chosen a more loosely coupled way to select features: a la carte construction from component technologies. Rather than using AFS for the global information service, Internet users chose from a wealth of session protocols (Gopher, HTTP, etc.), presentation-layer services (Kerberos, PGP, Lempel-Ziv compression, etc.), and separate cache and replication services. At present this has lead to some poor choices (e.g., running the Web without caching support), but economics will push the Internet into a better technical configuration in the not-too-distant future. Moreover, in a rapidly changing, competitive multi-vendor environment it is more realistic to combine features from component technologies than to wrap a “complete” set in an operating system.

6 Related Efforts

There has been a great deal of research into caching. We restrict our discussion here to wide-area network caching efforts.

One of the earliest efforts to support caching in a wide-area network environment was the Domain Naming System [16]. While not a general file or object cache, the DNS supports caching of name lookup results from server to server

and also from client to server (although the widespread BIND *resolver* client library does not provide client caching), using timeouts for cache consistency.

AFS provides a wide-area file system environment, supporting whole file caching [13]. Unlike the Harvest cache, AFS handles cache consistency using a server callback scheme that exhibits scaling problems in an environment where objects can be globally popular. The Harvest cache implementation we currently make available uses timeouts for cache consistency, but we also experimented with a hierarchical invalidation scheme (see Section 4). Also, Harvest implements a more general caching interface, allowing objects to be cached using a variety of access protocols (FTP, Gopher, and HTTP), while AFS only caches using the single AFS access protocol.

Gwertzman and Seltzer investigated a mechanism called *geographical push caching* [12], in which the server chooses to replicate documents as a function of observed traffic patterns. That approach has the advantage that the choice of what to cache and where to place copies can be made using the server’s global knowledge of reference behavior. In contrast, Bestavros et al. [11] explored the idea of letting clients make the choice about what to cache, based on application-level knowledge such as user profiles and locally configured descriptions of organizational boundaries. Their choice was motivated by their finding that cache performance could be improved by biasing the cache replacement policy in favor of more heavily shared local documents. Bestavros also explored a mechanism for distributing popular documents based on server knowledge [3].

There have also been a number of simulation studies of caching in large environments. Using trace-driven simulations Alonso and Blaze showed that server load could be reduced by 60-90% [1, 2]. Muntz and Honeyman showed that a caching hierarchy does not help for typical UNIX workloads [17]. A few years ago, we demonstrated that FTP access patterns exhibit significant sharing and calculated that as early as 1992, 30-50% of NSFNET traffic was caused by repeated access to read-only FTP objects [10].

There have also been several network object cache implementations, including the CERN cache [15], Lagoon [6], and the Netscape client cache. Netscape currently uses a 5 MB default disk cache at each client, which can improve client performance, but a single user might not have a high enough hit rate to affect network traffic substantially. Both the CERN cache and Lagoon effort improve client performance by allowing alternate access points for heavily popular objects. Compared to a client cache, this has the additional benefit of distributing traffic, but the approach (forking server) lacks required scalability. Harvest is unique among these systems in its support for a caching hierarchy, and in its high performance implementation. Its hierarchical approach distributes and reduces traffic, and the non-blocking/non-forking architecture provides greater scalability. It can be used to increase server performance, client performance, or both.

Cate’s *Alex* file system [8], completed before the explosive growth of the Web, exports a cache of anonymous FTP space via an NFS interface. For performance, Alex caches IP addresses, keeps object meta-data in memory, and caches FTP connections to remote servers to stream fetches to multiple files. Alex uses TTL-based consistency, caching files for one tenth of the elapsed time between the file was fetched and the file’s creation/modification date. The architecture of the Harvest cache is similar to Alex in many ways: Harvest caches IP addresses, keeps meta-data in memory, and

implements a similar life-time based object consistency algorithm. Harvest does not stream connections to Gopher and Web servers, because these protocols do not yet support streaming access. In contrast to Alex, which exports FTP files via the UDP-based NFS protocol, Harvest exports Gopher, FTP, and Web objects via the proxy-http interface implemented by Web browsers. Furthermore, the Harvest cache supports hierarchical caching, implements a consistency protocol tailored for Web objects, and serves as a very fast httpd-accelerator.

7 Summary

Internet information systems have evolved so rapidly that they postponed performance and scalability for the sake of functionality and easy deployment. However, they cannot continue to meet exponentially growing demand without new infrastructure. Towards this end, we designed the Harvest hierarchical object cache.

This paper presents measurements that show that the Harvest cache achieves better than an order of magnitude performance improvement over other proxy caches. It also demonstrates that HTTP is *not* an inherently slow protocol, but rather that many popular implementations have ignored the sage advice to make the common case fast [14].

Hierarchical caching distributes load away from server hot spots raised by globally popular information objects, reduces access latency, and protects the network from erroneous clients. High performance is particularly important for higher levels in the cache hierarchy, which may experience heavy service request rates.

The Internet's autonomy and scale present difficult challenges to the way we design and build system software. Once software becomes accepted as de facto standards, both its merits and its deficiencies may live forever. For this reason, the real-world complexities of the Internet make one face difficult design decisions. The maze of protocols, independent software implementations, and well-known bugs that comprise the Internet's upper layers, frequently force tradeoffs between design cleanliness and operational transparency. This paper discusses many of the tradeoffs forced upon us.

Software and Measurement Data Availability

The Harvest cache runs under several operating systems, including SunOS, Solaris, DEC OSF-1, HP/UX, SGI, Linux, and IBM AIX. Binary and source distributions of the cache are available from <http://excalibur.usc.edu/>. The test code and the list of URLs employed in the performance evaluation presented here are available from <http://excalibur.usc.edu/experiments>. General information about the Harvest system, including the cache user's manual, is available from <http://harvest.cs.colorado.edu/>

Acknowledgments

This work was supported in part by the Advanced Research Projects Agency under contract number DABT63-93-C-0052. Danzig was also supported in part by the Air Force Office of Scientific Research under Award Number F49620-93-1-0082, and by a grant from Hughes Aircraft Company under NASA EOSDIS project subcontract number ECS-00009, and by National Science Foundation Institutional Infrastructure Grant Number CDA-921632. Schwartz was also supported in part by the National Science Foundation under grant numbers NCR-9105372 and NCR-9204853, an equipment grant from Sun Microsystems' Collaborative Research Program,

and from the University of Colorado's Office of the Vice Chancellor for Academic Affairs. Chuck Neerdaels was supported by HBP NIH grant 1-P20-MH/DA52194-01A1.

The information contained in this paper does not necessarily reflect the position or the policy of the U.S. Government or other sponsors of this research. No official endorsement should be inferred.

We thank John Noll for writing the initial cache prototype. We thank Darren Hardy, Erhyuan Tsai, and Duane Wessels for all of the work they have done on integrating the cache into the overall Harvest system.

References

- [1] Rafael Alonso and Matthew Blaze. Long-term caching strategies for very large distributed file systems. *Proceedings of the USENIX Summer Conference*, pages 3–16, June 1991.
- [2] Rafael Alonso and Matthew Blaze. Dynamic hierarchical caching for large-scale distributed file systems. *Proceedings of the Twelfth International Conference on Distributed Computing Systems*, June 1992.
- [3] Azer Bestavros. *Demand-Based Document Dissemination for the World-Wide Web*. Computer Science Department, Boston University, February 1995. Available from <ftp://cs-ftp.bu.edu/techreports/95-003-web-server-dissemination.ps.Z>.
- [4] Nathaniel Borenstein and Ned Freed. RFC 1521: MIME (Multipurpose Internet Mail Extensions) part one: Mechanisms for specifying and describing the format of Internet message bodies, September 1993.
- [5] C. Mic Bowman, Peter B. Danzig, Darren R. Hardy, Udi Manber, and Michael F. Schwartz. The Harvest information discovery and access system. *Proceedings of the Second International World Wide Web Conference*, pages 763–771, October 1994. Available from <ftp://ftp.cs.colorado.edu/pub/cs/techreports-schwartz/Harvest.Conf.ps.Z>.
- [6] Paul M. E. De Bra and Reiner D. J. Post. *Information Retrieval in the World-Wide Web: Making client-based searching feasible*. Available from <http://www.win.tue.nl/win/cs/is/reinpost/www94/www94.html>.
- [7] Hans-Werner Braun and Kimberly Claffy. Web traffic characterization: an assessment of the impact of caching documents from NCSA's web server. In *Second International World Wide Web Conference*, October 1994.
- [8] Vincent Cate. Alex - a global filesystem. *Proceedings of the Usenix File Systems Workshop*, pages 1–11, May 1992.
- [9] Peter B. Danzig, Katia Obraczka, and Anant Kumar. An analysis of wide-area name server traffic: A study of the Domain Name System. *ACM SIGCOMM 92 Conference*, pages 281–292, August 1992.
- [10] Peter B. Danzig, Michael F. Schwartz, and Richard S. Hall. A case for caching file objects inside internetworks. *ACM SIGCOMM 93 Conference*, pages 239–248, September 1993.

- [11] Bestavros et al. Application-level document caching in the Internet. *Workshop on Services in Distributed and Networked Environments, Summer 1995*. Available from <ftp://cs-ftp.bu.edu/techreports/95-002-web-client-caching.ps.Z>, January 1995.
- [12] James Gwertzman and Margo Seltzer. The case for geographical push-caching. *HotOS Conference*, 1994. Available as <ftp://das-ftp.harvard.edu/techreports/tr-34-94.ps.gz>.
- [13] John Howard, Michael Kazar, Sherri Menees, David Nichols, M. Satyanarayanan, Robert Sidebotham, and Michael West. Scale and performance in a distributed file system. *ACM Transactions on Computer Systems*, 6(1):51–81, February 1988.
- [14] Butler Lampson. Hints for computer system design. *Operating Systems Review*, 17(5):33–48, Oct 10-13, 1983.
- [15] Ari Luotonen, Henrik Frystyk, and Tim Berners-Lee. CERN HTTPD public domain full-featured hypertext/proxy server with caching, 1994. Available from <http://info.cern.ch/hypertext/WWW/Daemon/Status.html>.
- [16] Paul Mockapetris. RFC 1035: Domain names - implementation and specification. Technical report, University of Southern California Information Sciences Institute, November 1987.
- [17] D. Muntz and P. Honeyman. Multi-level caching in distributed file systems - or - your cache ain't nuthin' but trash. *Proceedings of the USENIX Winter Conference*, pages 305–313, January 1992.
- [18] Mirjana Spasojevic, Mic Bowman, and Alfred Spector. *Information Sharing Over a Wide-Area File System*. Transarc Corporation, July 1994. Available from <ftp://grand.central.org/darpa/arpa2/papers/usenix95.ps>.
- [19] Kurt Jeffery Worrell. *Invalidation in Large Scale Network Object Caches*. Department of Computer Science, University of Colorado, Boulder, Colorado, December 1994. M.S. Thesis, available from <ftp://ftp.cs.colorado.edu/pub/cs/techreports/schwartz/WorrellThesis.ps.Z>.

Anawat Chankhunthod received his B.Eng in Electrical Engineering from the Chiang Mai University, Thailand in 1991 and his M.S in Computer Engineering from the University of Southern California in 1994. He is currently a Ph.D candidate in Computer Engineering at the University of Southern California. Shortly after receiving his B.Eng, he joined the faculty of the Department of Electrical Engineering, Chiang Mai University and currently is on leave for extending his education. He is currently an research assistant at the Networking and Distributed system laboratory at the University of Southern California. His research focuses on computer networking and distributed systems. He can be contacted at chankhun@usc.edu.

Peter B. Danzig received his B.S. in Applied Physics from the University of California Davis in 1982 and his Ph.D in Computer Science from the University of California Berkeley in 1989. He is currently an Assistant Professor at

the University of Southern California. His research addresses both building scalable Internet information systems and flow, congestion and admission control algorithms for the Internet. He has served on several ACM SIGCOMM and ACM SIGMETRICS program committees and is an associate editor of *Internetworking: Research and Experience*. He can be contacted at danzig@usc.edu.

Charles J. Neerdaels received his BAEM in Aerospace Engineering and Mechanics in 1989, from the University of Minnesota, Minneapolis. After several years work in the defense industry, he continued his education in Computer Science at the University of Southern California. He has recently left the University to become a Member of Technical Staff, Proxy Development at Netscape communications, and can be reached at chuckn@netscape.com. CA 94043

Michael Schwartz received his Ph.D in Computer Science from the University of Washington in 1987, after which time he joined the faculty of the Computer Science Department at the University of Colorado - Boulder. Schwartz' research focuses on international-scale networks and distributed systems. He has built and experimented with a dozen information systems, and chairs the IRTF Research Group on Resource Discovery, which built the the *Harvest* system. Schwartz is on the editorial board for *IEEE/ACM Transactions on Networking*, and was a guest editor of *IEEE Journal of Selected Areas in Communication*, for a 1995 Special Issue on the Global Internet. In 1995 Schwartz joined @Home (a Silicon Valley startup doing Internet over cable), where he is currently leading the directory service effort. Schwartz can be reached at schwartz@home.net.

Tracking and Viewing Changes on the Web

Fred Dougis Thomas Ball
AT&T Bell Laboratories

Abstract

We describe a set of tools that detect when World-Wide-Web pages have been modified and present the modifications visually to the user through marked-up HTML. The tools consist of three components: *w3newer*, which detects changes to pages; *snapshot*, which permits a user to store a copy of an arbitrary Web page and to compare any subsequent version of a page with the saved version; and *HtmlDiff*, which marks up HTML text to indicate how it has changed from a previous version. We refer to the tools collectively as the *AT&T Internet Difference Engine* (AIDE). This paper discusses several aspects of AIDE, with an emphasis on systems issues such as scalability, security, and error conditions.

1 Introduction

Use of the World-Wide-Web (W^3) has increased dramatically over the past couple of years, both in the volume of traffic and the variety of users and content providers. The W^3 has become an information distribution medium for academic environments (its original motivation), commercial ones, and virtual communities of people who share interests in a wide variety of topics. Information that used to be sent out over electronic mail or USENET, both active media that go to users who have subscribed to mailing lists or newsgroups, can now be posted on a W^3 page. Users interested in that data then visit the page to get the new information.

The URLs of pages of interest to a user can be saved in a “hotlist” (known as a bookmark file in Netscape), so they can be visited conveniently. How does a user find out when pages have changed? If users know that pages contain up-to-the-minute data (such as stock quotes), or are frequently changed by their owners, they may visit the pages often. Other pages may be ignored, or browsed by the user only to find they have not changed.

In recent months, several tools have become available to address the problem of determining when a page has changed. The tool with perhaps the widest distribu-

tion is “Smart Bookmarks,” from First Floor Software, which has been incorporated into Netscape for Windows as “Netscape SmartMarks.” Users have items in their bookmark list automatically polled to determine if they have been modified. In addition, content providers can optionally embed *bulletins* in their pages, which allow short messages about a page to be displayed in a page that refers to it. Other tools for learning about modifications are discussed in the next section.

Each of the current tools suffers from a significant deficiency: while they provide the user with the knowledge that the page has changed, they show little or no information about *how* the page has changed. Although a few pages are edited by their maintainers to highlight the most recent changes, often the modifications are not prominent, especially if the pages are large. Even pages with special highlighting of recent changes (including bulletins) are problematic: if a user visits a page frequently, what is “new” to the maintainer may not be “new” to the user. Alternatively, a user who visits a page infrequently may miss changes that the maintainer deems to be old. Changes deemed uninteresting by a document’s author and omitted from a change summary or bulletin actually may be of great interest to readers. Finally, the really major change might be the item that was deleted or modified, rather than added. Such items are unlikely to be found on a “What’s New?” page.

We have developed a system that efficiently tracks when pages change, compactly stores versions on a per-user basis, and automatically compares and presents the differences between pages. The *AT&T Internet Difference Engine* (AIDE) provides “personalized” views of versions of W^3 pages with three tools. The first, *w3newer*, is a derivative of one of the existing modification tracking tools, *w3new*, discussed in the next section. It periodically accesses the W^3 to find when pages on a user’s hotlist have changed. The second, *snapshot*, allows a user to save versions of a page and later use a third tool, *HtmlDiff*, to see how it has changed. *HtmlDiff* automatically compares two HTML pages and creates a “merged” page to show the differences with special HTML markups.

While AIDE can help arbitrary users track pages of interest, it can be of particular use in a collaborative environment. One example of a collaborative environment on the W^3 is the *WikiWikiWeb* [3], which allows multiple users to edit the content of documents dynamically. There is a RecentChanges page that sorts documents by modification date. Typically content is added to the end of a page and it is not difficult to determine visually what changes occurred since the last visit. However, content can be modified anywhere on the page, and those changes may be too subtle to notice. Within AT&T, a clone of *WikiWikiWeb*, called *WebWeaver*, stores its own version archive and uses *HtmlDiff* to show users the differences from earlier versions of a page. While the differences are not currently customized for each user, that would be a natural and simple extension. Similarly, integrating the RecentChanges page into the AIDE report of what's new would avoid having to query multiple sources to determine what has recently changed.

The rest of this paper is organized as follows. Section 2 describes related work. Section 3 elaborates on our extensions to *w3new*, Section 4 describes the *snapshot* versioning tool, and Section 5 describes *HtmlDiff* in detail. Section 6 describes the integration of the three tools into AIDE. Section 7 relates early experiences with the system. Section 8 discusses other extensions and uses of AIDE, and Section 9 concludes.

2 Related Work

2.1 Tracking Modifications

There has been a great deal of interest lately in finding out when pages on the W^3 have changed. As mentioned above, Smart Bookmarks checks for modifications and also supports an extension to HTML to allow a description of a page, or recent changes to it, to be obtained along with other "header" information such as the last modification date. These bulletins may be useful in some cases but will not help in others. For instance, W^3 Virtual Library pages contain many links to other pages within some subject area and have a number of links added at a time; a bulletin that announces that "10 new links have been added" will not point the user to the specific locations in the page that have been edited. Also, it suffers from the problem of timeliness mentioned in the introduction.

Smart Bookmarks have the advantage of being integrated directly with a user's hotlist, making a visual indication of what has changed available without resorting to a separate page. Several other tools read a user's hotlist and generate a report, in HTML, of recently changed pages. Examples include *webwatch* [16], a

product for Windows; *w3new* [4], a public-domain Perl script that runs on UNIX; and *Katipo* [13], which runs on the Macintosh.

Another similar tool, *URL-minder* [19], runs as a service on the W^3 itself and sends email when a page changes. Unlike the tools that run on the user's host and use the "hotlist" to determine which URLs to check, *URL-minder* acts on URLs provided explicitly by a user via an HTML form. Centralizing the update checks on a W^3 server has the advantage of polling hosts only once regardless of the number of users interested. However, the need to send URLs explicitly through a form is cumbersome.

There are two basic strategies for deciding when a page has changed. Most tools use the HTTP HEAD command to retrieve the Last-Modified field from a W^3 document, either returning a sorted list of all modification times or just those times that are different from the browser's history (the timestamp of the version the user presumably last saw). *URL-minder* uses a checksum of the content of a page, so it can detect changes in pages that do not provide a Last-Modified date, such as output from Common Gateway Interface (CGI) scripts. *W3new* (and therefore *w3newer*) requests the Last-Modified date if available; otherwise, it retrieves and checksums the whole page.

These tools also vary with respect to frequency of checking and where the checks are performed. Most of the tools automatically run periodically from the user's machine. All URLs are checked each time the tools run, with the possible exception of *URL-minder*, which runs on an Internet server and checks pages with an arbitrary frequency that is guaranteed to be at least as often as some threshold, such as a week. (*URL-minder*'s implementation is hidden behind a CGI interface.)

2.2 Version Repositories

As discussed below, we use the Revision Control System (RCS) [18] to compactly maintain a history of documents, addressed by their URLs. A CGI interface to RCS allows a user to request a URL at a particular date, from anywhere on the W^3 . This is similar in spirit to the "time travel" capability of file systems such as 3DFS [5] that transparently allow access to files at arbitrary dates. A slight twist on the versioning is that we wish to track the times at which each user checked in a page, even if the page hasn't changed between check-ins of that page by different users. This is accomplished outside of RCS by maintaining a per-user control file, allowing quick access to a user's access history.

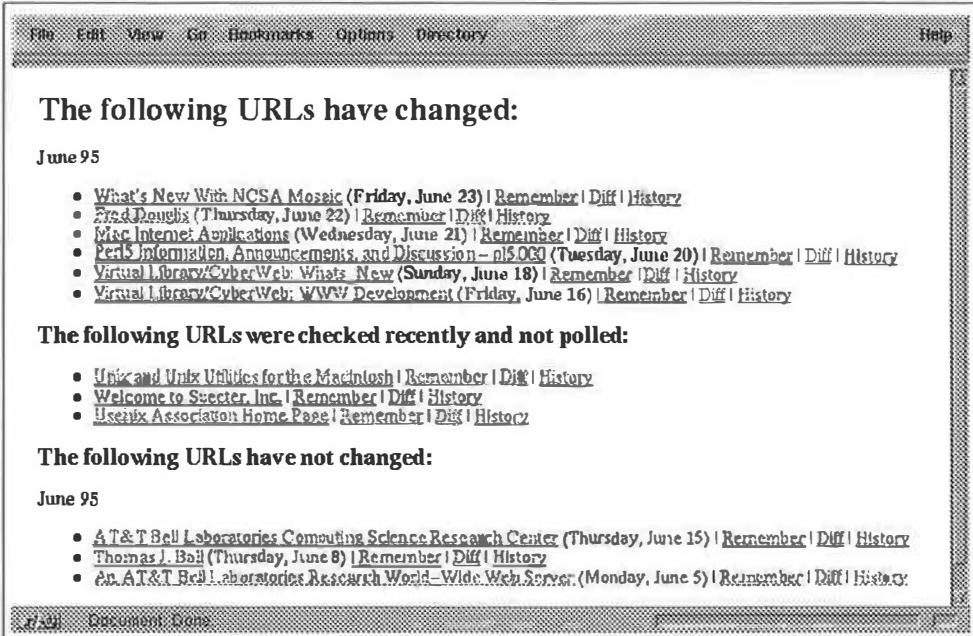


Figure 1: Output of *w3newer* showing a number of anchors (the descriptive text comes from the hotlist). The ones that are marked as “changed” have modification dates after the time the user’s browser history indicates the URL was seen. Some URLs were not checked at all, and others were checked and are known to have been seen by the user.

2.3 HTML Differencing

We know of no existing tools that compare HTML pages and present the comparison as HTML. However, there is much closely related work on heuristics for parallel document alignment and similarity measures between documents [2] that we benefit from. Line-based comparison utilities such as UNIX *diff* [10] clearly are ill-suited to the comparison of structured documents such as HTML. Most modern word processing programs have a “revision” mode that track additions and deletions on-line as an author modifies a document, graphically annotating the differences. *Html-Diff* graphically annotates differences in a similar manner but operates off-line after document creation, using heuristics to determine the additions and deletions.

3 Tracking Modifications

To our knowledge, the tools described in Section 2.1 poll every URL with the same frequency. We modified *w3newer*¹ to make it more scalable, as well as to integrate it with the other components of AIDE. Figure 1

¹ *W3newer* was selected because it ran on UNIX and because it was available before the others. If Smart Bookmarks had been available at the time, it would have provided a better starting point.

gives an example of the report generated by *w3newer*; the meaning of the links on the right-hand side is discussed in Section 6.

W3newer runs on the user’s machine, but it omits checks of pages already known to be modified since the user last saw the page, and pages that have been viewed by the user within some threshold. The time when the user has viewed the page comes from the *W³* browser’s history. The “known modification date” comes from a variety of sources: a cached modification date from previous runs of *w3newer*; a modification date stored in a proxy-caching server’s cache; or the HEAD information provided by *httpd* (the HTTP server) for the URL. If either of the first two sources of the modification date indicate that the page has not been visited since it was modified, then HTTP is used only if the time the modification information was obtained was long enough ago to be considered “stale” (currently, the threshold is one week).

In addition, there is a threshold associated with each page to determine the maximum frequency of direct HEAD requests. If the page was visited within the threshold, or the modification date obtained from the proxy-caching server is current with respect to the threshold, the page is not checked. The threshold can vary depending on the URL, with *perl* pattern match-

# Comments start with a sharp sign.	
# perl syntax requires that “.” be escaped	
# Default is equivalent to ending the file with “.*”	
Default	2d
file:*	0
http://www.yahoo.com/*	7d
http://*.att.com/*	0
http://www.ncsa.miiuc.edu/SDG/Software/- 12h Mosaic/Docs/whats-new\.html	
http://snapple.cs.washington.edu:600/- mobile/	1d
# this is in my hotlist but will be different every day	
http://www.unitedmedia.com/- comics/dilbert/	never

Table 1: An example of the thresholds specified to *w3newer*. The table is explained in the text.

ing used to determine what threshold to apply. The first matching pattern is used.

Table 1 gives an example of a *w3newer* configuration file. Thresholds are specified as combinations of days (d) and hours (h), with 0 indicating that a page should be checked on every run of *w3newer* and never indicating that it should never be checked. In this example, URLs are checked every two days by default, but some URLs are overridden. Local files are checked upon every execution, since a *stat* call is cheap, and anything in the *att.com* domain is checked upon every execution as well. Things on *Yahoo* are checked only every seven days in order to reduce unnecessary load on that server, since the user doesn’t expect to revisit *Yahoo* pages daily even if they change. *Dilbert* is never checked because it will always be different.

3.1 System Issues

Cache Consistency

Determining when HTTP pages have changed is analogous to caching a file in a distributed file system and determining when the file has been modified. While file systems such as the Andrew File System [9] and Sprite [12] provide guarantees of cache consistency by issuing call-backs to hosts with invalid copies, HTTP access is closer to the traditional NFS approach, in which clients check back with servers periodically for each file they access. Netscape can be configured to check the modification date of a cached page each time it is visited, once each session, or not at all. Caching servers check when a client forces a full reload, or after a time-to-live value expires.

Here the problem is complicated by the target envi-

ronment: one wishes to know not only when a currently viewed page has changes, but also when a page that has not been seen in a while has changed. Fortunately, unlike with file systems, HTTP data can usually tolerate some inconsistency. In the case of pages that are of interest to a user but have not been seen recently, finding out within some reasonable period of time, such as a day or a week, will usually suffice. Even if servers had a mechanism to notify all interested parties when a page has changed, immediate notification might not be worth the overhead.

Instead, one could envision using something like the Harvest replication and caching services [1] to notify interested parties in a lazy fashion. A user who expresses an interest in a page, or a browser that is currently caching a page, could register an interest in the page with its local caching service. The caching service would in turn register an interest with an Internet-wide, distributed service that would make a best effort to notify the caching service of changes in a timely fashion. (This service could potentially archive versions of HTTP pages as well.) Pages would already be replicated, with server load distributed, and the mechanism for discovering when a page changes could be left to a negotiation between the distributed repository and the content provider: either the content provider notifies the repository of changes, or the repository polls it periodically. Either way, there would not be a large number of clients polling each interesting HTTP server. Moving intelligence about HTTP caching to the server has been proposed by Gwertzman and Seltzer [7] and others.

One could also envision integrating the functionality of AIDE into file systems. Tools that can take actions when arbitrary files change are not widely available, though they do exist [15]. Users might like to have a unified report of new files and *W*³ pages, and *w3newer* supports the “file:” specification and can find out if a local file has changed. However, *snapshot* has no way to access a file on the user’s (remote) file system. Moving functionality into the browser would allow individual users to take snapshots of files that are not already under the control of a versioning system such as RCS; this might be an appropriate use of a browser with client-side execution, such as *HotJava* [17] or recent versions of Netscape.

Error Conditions

When a periodic task checks the status of a large number of URLs, a number of things can go wrong. Local problems such as network connectivity or the status of a proxy-caching server can cause *all* HTTP requests to fail. Proxy-caching servers are sometimes overloaded

to the point of timing out large numbers of requests, and a background task that retrieves many URLs in a short time can aggravate their condition. *W3newer* should therefore be able to detect cases when it should abort and try again later (preferably in time for the user to see an updated report).

At the same time, a number of errors can arise with individual URLs. A URL can move, with or without leaving a forwarding pointer. The server for a URL can be deactivated or renamed. Its site may disallow retrieval of this URL by “robots,” meaning that the administrator for its site has created a special file, robots.txt, and requested that automated programs such as “web crawlers” not retrieve the URL. Currently, programs only voluntarily follow the “robot exclusion protocol” [14], the convention that defines the use of robots.txt. Although *w3newer* currently obeys this protocol, it is not clear that it should, at least for URLs the user would otherwise access directly periodically.

Finally, automatic detection of modifications based on information such as modification date and checksum can lead to the generation of “junk mail” as “noisy” modifications trigger change notifications. For instance, pages that report the number of times they have been accessed, or embed the current time, will look different every time they are retrieved.

W3newer attempts to address these issues by the following steps:

- If a URL is inaccessible to robots, that fact is cached so the page is not accessed again unless a special flag is set when the script is invoked.
- Another flag can tell *w3newer* to treat error conditions as a successful check as far as the URL’s timestamp goes, so that a URL with some problem will be checked with the same frequency as an accessible one. In general, however, it seems that errors are likely to be transient, and checking the next time *w3newer* is run is reasonable.
- When a URL is inaccessible, an error message appears in the status report, so the user can take action to remove a URL that no longer exists or repeatedly hits errors.

In addition, *w3newer* could be modified to keep a running counter of the number of times an error is encountered for a particular URL, or to skip subsequent URLs for a host if a host or network error (such as “timeout” or “network unreachable”) has already occurred. Addressing the problem of “noisy” modifications will require heuristics to examine the differences at a semantic level.

4 Snapshots

In addition to providing a mechanism for determining when W^3 pages have been modified, there must be a way to access multiple versions of a page for the purposes of comparison. This section describes methods for maintaining version histories and several issues that arise from our solution.

4.1 Alternative Approaches

There are three possible approaches for providing versioning of W^3 pages: making each content provider keep a history of all versions, making each user keep this history, or storing the version histories on an external server.

Server-side support Each server could store a history of its pages and provide a mechanism to use that history to produce marked-up pages that highlight changes. This method requires arbitrary content providers to provide versioning and differencing, so it is not practical, although it is desirable to support this feature when the content provider is willing. (See Section 8.1.)

Client-side support Each user could run a program that would store items in the hotlist locally, and run *HtmlDiff* against a locally saved copy. This method requires that every page of interest be saved by every user, which is unattractive as the number of pages in the average user’s hotlist increases, and it also requires the ability to run *HtmlDiff* on every platform that runs a W^3 browser. Storing the pages referenced by the hotlist may not be too unreasonable, since programs like Netscape may cache pages locally anyway. There are other external tools such as *warm-list* [20] that provide this functionality.

External service Our approach is to run a service that is separate from both the content provider and the client, and uses RCS to store versions. Pages can be registered with the service via an HTML form, and differences can be retrieved in the same fashion. Once a page is stored with the service, subsequent requests to remember the state of the page result in an RCS “check-in” operation that saves only the differences between the page and its previously checked-in version. Thus, except for pages that change in many respects at once, the storage overhead is minimal beyond the need to save a copy of the page in the first place.

Drawbacks to the “external service” approach are that the service must remember the state of every page

that anyone who uses the service has indicated an interest in and must know which user has seen which version of each page. The first issue is primarily one of resource allocation, and is not expected to be a significant issue unless the service is used by a great many clients on a number of large pages. The second issue is addressed in our initial prototype by using RCS's support for timestamps and requesting a page as it existed at a particular time. In the next version of the system, a set of version numbers is retained for each <user, URL> combination. This removes any confusion that could arise if the timestamps provided for a page do not increase monotonically and also makes it easier to present the user with a set of versions seen by that person regardless of what other versions are also stored.

Relative links become a problem when a page is moved away from the machine that originally provided it. If the source were passed along unmodified, then the *W*³ browser would consider links to be relative to the CGI directory containing the *snapshot* script. HTML supports a BASE directive that makes relative links relative to a different URL, which mostly addresses this problem; however, Netscape 1.1N treats internal links within such a document to be relative to the new BASE as well, which can cause the browser to jump between the *HtmlDiff* output and the original document unexpectedly.

4.2 System Issues

The *snapshot* facility must address four important issues: use of CGI, synchronization, resource utilization, and security/privacy.

CGI is a problem because there is no way for *snapshot* to interact with the user and the user's browser, other than by sending HTML output. (The system does not currently assume the ability of a browser to support Java [11], although moving to Java in the future is possible and might help address some of these issues.) When a CGI script is invoked, *httpd* sets up a default timeout, and if the script does not generate output for a full timeout interval, *httpd* will return an error to the browser. This was a problem for *snapshot* because the script might have to retrieve a page over the Internet and then do a time-consuming comparison against an archived version. The server does not tell *snapshot* what a reasonable timeout interval might be for any subsequent retrievals; instead this is hard-coded into the script. In order to keep the HTTP connection alive, *snapshot* forks a child process that generates one space character (ignored by the *W*³ browser) every several seconds while the parent is retrieving a page or executing *HtmlDiff*.

Synchronization between simultaneous users of the facility is complicated by the use of multiple files for bookkeeping. The system must synchronize access to the RCS repository, the locally cached copy of the HTML document, and the control files that record the versions of each page a user has checked in. Currently this is done by using UNIX file locking on both a per-URL lock file and the per-user control file. Ideally the locks could be queued such that if multiple users request the same page simultaneously, the second *snapshot* process would just wait for the page and then return, rather than repeating the work. This is not so important for making snapshots, in which case a proxy-caching server can respond to the second request quickly and RCS can easily determine that nothing has changed, but there is no reason to run *HtmlDiff* twice on the same data.

The latter point relates to the general issue of resource utilization. *Snapshot* has the potential to use large amounts of both processing and disk space. The need to execute *HtmlDiff* on the server can result in high processor loads if the facility is heavily used. These loads can be alleviated by caching the output of *HtmlDiff* for a while, so many users who have seen versions *N* and *N* + 1 of a page could retrieve *HtmlDiff*(*page_N*, *page_{N+1}*) with a single invocation of *HtmlDiff*. The facility could also impose a limit on the number of simultaneous users, or replicate itself among multiple computers, as many *W*³ services do.

Lastly, security and privacy are important. Because the CGI scripts run with minimal privileges, from an account to which many people have access, the data in the repository is vulnerable to any CGI script and any user with access to the CGI area. Data in this repository can be browsed, altered, or deleted. In order to use the facility one must give an identifier (currently one's email address, which anyone can specify) that is used subsequently to compare version numbers. Browsing the repository can therefore indicate which user has an interest in which page, how often the user has saved a new checkpoint, and so on.

By moving to an authenticated system on a secure machine, one could break some of these connections and obscure individuals' activities while providing better security. The repository would associate impersonal account identifiers with a set of URLs and version numbers, and passwords would be needed to access one of these accounts. Whoever administers this facility, however, will still have information about which user accesses which pages, unless the account creation can be done anonymously.

5 Comparison of HTML pages

In our experience, only a small fraction of pages on the W^3 contain information that allows users to ascertain how the pages have changed—examples include icons that highlight recent additions, a link to a “changelog”, or a special “What’s New” page. As was mentioned in the introduction, these approaches suffer from deficiencies. They are intended to be viewed by all users, but users will visit the pages at different intervals and have different ideas of “what’s new”. In addition, the maintainer must explicitly generate the list of recent changes, usually by manually marking up the HTML.

Automatic comparison of HTML pages and generation of marked-up pages frees the HTML provider from having to determine what’s new and creating new or modified HTML pages to point to the differences. There are many ways to compare documents and many ways to present the results. This section describes various models for the comparison of HTML documents, our comparison algorithm, and issues involved in presenting the results of the comparison.

5.1 What’s in a Diff?

HTML separates content (raw text) from markups. While many markups (such as `<P>`, `<I>`, and `<HR>`) simply change the formatting and presentation of the raw text, certain markups such as images (``) and hypertext references (``) are “content-defining.” Whitespace in a document does not provide any content (except perhaps inside a `<PRE>`), and should not affect comparison.

At one extreme, one can view an HTML document as merely a sequence of words and “content-defining” markups. Markups that are not “content-defining” as well as whitespace are ignored for the purposes of comparison. The fact that the text inside `<P>...</P>` is logically grouped together as a paragraph is lost. As a result, if one took the text of a paragraph comprised of four sentences and turned it into a list (``) of four sentences (each starting with ``), no difference would be flagged because the content matches exactly.

At the other extreme, one can view HTML as a hierarchical document and compare the parse tree or abstract syntax tree representations of the documents, using subtree equality (or some weaker measure) as a basis for comparison. In this case, a subtree representing a paragraph (`<P>...</P>`) might be incomparable with a subtree representing a list (`...`). The example of replacing a paragraph with a list would be flagged as both a content and format change.

We view an HTML document as a sequence of sentences and “sentence-breaking” markups (such as `<P>`, `<HR>`, ``, or `<H1>`) where a “sentence” is a sequence of words and certain (non-sentence-breaking) markups (such as `` or `<A>`). A “sentence” contains at most one English sentence, but may be a fragment of an English sentence. All markups are represented and are compared, regardless of whether or not those markups are “content-defining” (however, as described later, certain markups may not be highlighted as having changed). In the paragraph-to-list example, the comparison would show no change to content, but a change to the formatting.

We apply Hirshberg’s solution to the longest common subsequence (LCS) problem [8] (with several speed optimizations) to compare HTML documents. This is the well-known comparison algorithm used by the UNIX `diff` utility [10]. The LCS problem is to find a (not necessarily contiguous) common subsequence of two sequences of tokens that has the longest length (or greatest weight). Tokens not in the LCS represent changes. In UNIX `diff`, a token is a textual line and each line has weight equal to 1. In *HtmlDiff*, a token is either a sentence-breaking markup or a sentence, which consists of a sequence of words and non-sentence-breaking markups. Note that the definition of sentence is *not* recursive; sentences cannot contain sentences. A simple lexical analysis of an HTML document creates the token sequence and converts the case of the markup name and associated (variable,value) pairs to uppercase; parsing is not required.

We now describe how the weighted LCS algorithm compares two tokens and computes a non-negative weight reflecting the degree to which they match (a weight of 0 denotes no match). Sentence-breaking markups can only match sentence-breaking markups. They must be identical (modulo whitespace, case, and reordering of (variable,value) pairs) in order to match (see section 5.3 for a discussion of the ramifications of this). A match has weight equal to 1. Sentences can match only sentences, but sentences need not be identical to match one another. We use two steps to determine whether or not two sentences match. The first step uses sentence length as a comparison metric. Sentence length is defined to be the number of words and “content-defining” markups such as `` or `<A>` in a sentence. Markups such as `` or `<I>` are not counted. If the lengths of two sentences are not “sufficiently close,” then they do not match. Otherwise, the second step computes the LCS of the two sentences (where words matching exactly against words are assigned weight 1, and markups match exactly against markups, as before). Let W be the number of words and content-defining markups in the LCS of the two

sentences and let L be the sum of the lengths of the two sentences. If the percentage $(2 * W)/L$ is sufficiently large, then the sentences match with weight W . Otherwise, they do not match.

5.2 Presentation of the differences

The comparison algorithm outlined above yields a mapping from the tokens of the old document to the tokens of the new document. Tokens that have a mapping are termed “common”; tokens that are in the old (new) document but have no counterpart in the new (old) are “old” (“new”). We refer to the “old” and “new” tokens as “differences”.

We investigated three basic ways to present the differences by creating HTML documents:

Side-by-Side A side-by-side presentation of the documents with common text vertically synchronized is a very popular and pleasing way to display the differences between documents (see, for example, UNIX *sdiff* or SGI’s graphical diff tool *gdiff*). Unfortunately, there is no good mechanism in place with current HTML and browser technology that allows such synchronization.

Only Differences Show only differences (old and new) and eliminate the common part (as done in UNIX *diff*). This optimizes for the “common” case, where there is much in common between the documents. This is especially useful for very large documents but can be confusing because of the loss of surrounding common context. Another problem with this approach is that an HTML document comprised of an interleaving of old and new fragments might be syntactically incorrect.

Merged-page Create an HTML page that summarizes all of the common, new, and old material. This has the advantage that the common material is displayed just once (unlike the side-by-side presentation). However, incorporating two pages into one again raises the danger of creating syntactically or semantically incorrect HTML (consider converting a list of items into a table, for example).

Our preference is to present the differences in the merged-page format to provide context and use internal hypertext references to link the differences together in a chain so the user can quickly jump from difference to difference. We currently deal with the syntactic/semantic problem of merging by eliminating all old markups from the merged page (note that this doesn’t mean all markups in the older document, just the ones

classified as “old” by the comparison algorithm). As a result, old hypertext references and images do not appear in the merged page (of course, since they were deleted they may not be accessible anyway). However, by reversing the sense of “old” and “new” one can create a merged page with the old markups intact and the new deleted. A more Draconian option would be to leave out all old material. In this case, there are no syntactic problems given that the most recent page is syntactically correct to begin with; the merged page is simply the most recent page plus some markups to point to the new material. We are exploring other ways to create a merged page.

An example of *HtmlDiff*’s merged-page output appears in Figure 2. Markups are used to highlight old and new material as follows. Two small arrow images are used to point to areas in the document that have changed. A red arrow points to old content and a green arrow points to new content. The arrows are also internal hypertext references to one another, linked in a chain to allow quick traversal of the differences. A banner at the front of the document contains a link to the first difference. Old text is displayed in “struck-out” font using *<STRIKE>*, which in our experience is rarely used in HTML found on the W^3 . Unfortunately, there is no ideal font for showing “new” text. We currently use *<I>*. Ideally, we would like to be able to color code the text to highlight, but this capability is not provided by all browsers.

Modified “content-defining” markups are highlighted, while changes to other markups (such as *<P>*) are not. Consider the example of changing the URL in an anchor but not the content surrounded by *<A>...*. In this case, an arrow will point to the text of the anchor, but the text itself will be in its original font.

5.3 Issues and Extensions

Since *HtmlDiff* can parse an HTML document and rectify certain syntactic problems, such as mismatched or missing markups, the only real problem it is likely to encounter is a set of changes that are so pervasive as to make the resulting merged HTML unreadable. For instance, if every other line were changed, then the mixture of unrelated struck-out and emphasized text would be muddled. We are experimenting with methods for varying the degree to which old and new text can be interspersed, as well as thresholds to specify when the changes are too numerous to display meaningfully.

Currently, *HtmlDiff* is neither “version-aware” nor “web-aware”. That is, *HtmlDiff* only compares the text of two HTML pages. It does not compare versions of the entities that the pages refer to, access them,

HtmlDiff: Here is the ~~first difference~~. There are 8 differences on this page. ▶ ~~is-old~~ ▶ *is new*.

USENIX
The Advanced Computing Systems Professional and Technical Association
SAGE
The System Administrators Guild

What's New

- ~~▶ Sale on USENIX Conference Proceedings~~
- Letter to SAGE Members about the "Gorton–Exon Amendment"
- ~~▶ Tenth System Administration Conference, September, 1996: Call for Papers~~ USENIX 1996 Technical Conference, January, 1996: ~~▶ Preliminary Program~~ ~~▶ Program & Registration~~
- ~~▶ Second Conference on Object-Oriented Technologies and Systems (COOTS), June 1996: Call for Papers~~
- Fourth Annual Tcl/Tk Workshop, July, 1996: Call for Papers
- Sixth UNIX Security Symposium, July, 1996: Call for Papers
- ~~▶ Tenth System Administration Conference, September, 1996: Call for Papers~~
- For USENIX members: Online Library of ~~▶ 1994–1995~~ ~~▶ 1993–1995~~ USENIX Proceedings

Figure 2: Output of *HtmlDiff* showing the differences between a subset of two versions of the USENIX Association home page (as of 9/29/95 and 11/3/95). Small arrows point to changes, with bold italics indicating additions and with deleted text struck out. The banner at the top of the page was inserted by *HtmlDiff*.

or invoke itself recursively on other referenced pages. This has a number of consequences. The good news is that *HtmlDiff* does not incur the overhead of pulling versions from a repository or sending requests over the *W³* for information. This cost is consumed by *w3newer* and *snapshot*. The bad news is that some differences may be ignored. For example, if the contents of an image file are changed but the URL of the file does not, then the URL in the page will not be flagged as changed. To support such comparison would require some sort of versioning of referenced entities and would also require *HtmlDiff* to have access to the version repositories. Full versioning of all entities would allow interesting comparisons to be done, but would dramatically increase storage requirements. A cheaper alternative would be to store a checksum of each entity and use the checksums to determine if something has changed. We are exploring how to efficiently perform such “smarter” comparisons.

6 Integrating the tools

There are two entry points to AIDE, one through *w3newer* and one through *snapshot*.

Currently, *w3newer* is invoked directly by the user, probably by a *cron* entry, and generates an HTML document indicating which pages have changed. As shown in Figure 1, *w3newer* associates three links with each document in the hotlist:

Remember Send the URL to the *snapshot* facility, to save a copy of the page. Though the page is retrieved, the RCS *ci* command ensures that it is not saved if it is unchanged from the previous time it was stored away.

Diff Have the *snapshot* facility invoke *HtmlDiff* to display the changes in a page since it was last saved away by the user.

History Have *snapshot* display a full log of versions

of this page, with the ability to run *HtmlDiff* on any pair of versions or to view a particular version directly.

Thus, each page that is reported as “new” can immediately be passed to *HtmlDiff*, and any page in the list can be “remembered” for future use. A user may also interface with *snapshot* directly, via a form, to check-in pages, view differences, or view the version history.

One disadvantage of the current approach is that there is no direct interaction between *w3newer*, *snapshot*, and the *W³* browser. Viewing a page with *HtmlDiff* does not cause the browser to record that the page has just been seen; instead, the browser records the URL that was used to invoke *HtmlDiff* in the first place. Subsequently, *w3newer* uses the obsolete timestamp from the browser and continues to report that the page has been modified more recently than the browser has seen it. As a result, the user must view a page directly as well as via *HtmlDiff* in order to both remove it from the list of modified pages and see the actual differences.

7 Experiences

In the approximately half-year since AIDE was built, we have been using the system regularly ourselves and have attempted to build a user community within AT&T. Personal use has been successful: one of us has recorded over 250 URLs and the other nearly 100. Adoption by others has been harder, and the reason we hear back from prospective users is nearly always the same: it is too time-consuming to install *w3newer* on one’s own machine. This reluctance is the primary motivation for moving the functionality of *w3newer* into the AIDE server.

In using AIDE ourselves, we realized another difficulty with the present arrangement: information overload. Merely sorting URLs by most recent modification dates is not satisfactory when the number of URLs grows into the hundreds. Instead, we are moving toward a user-specified prioritization of URLs along the lines of the Tapestry system, which prioritizes email and NetNews automatically [6].

So far, disk usage has not been a problem. There are over 500 URLs archived (many of these are for fixed collections of pages as described below in Section 8.2), and the archive uses under 8 Mbytes of disk storage (an average of 14.3 Kbytes/URL). Three files account for 2.7 Mbytes of that total, and each file is a URL that changes every 1–3 days and is being automatically archived upon each change.

8 Extensions

This section describes some possible extensions to the work already presented. Sections 8.1 and 8.2 discuss some interfaces that are already implemented, while Sections 8.3 and 8.4 presents unimplemented extensions to integrate tracking modifications into the server and to invoke scripts via the HTTP POST protocol.

8.1 Server-side Version Control

The tools described above do not require any changes to arbitrary servers or clients on the *W³*. Existing GET and POST protocols are used to communicate with specific servers that save versions of documents and provide marked-up versions showing how they have changed. However, if a server runs *HtmlDiff* and some *perl* scripts, it can provide a direct version-control interface and avoid the need to store copies of its HTML documents elsewhere.

The *perl* scripts we have written provide an interface to RCS [18]. A CGI script (/cgi-bin/rlog) converts the output of *rlog* into HTML, showing the user a history of the document with links to view any specific version or to see the differences between two versions. Another script (/cgi-bin/co) displays a version of a document under RCS control, while still another (/cgi-bin/rcsdiff) displays the differences. If the file’s name ends in .html then *HtmlDiff* is used to display the differences, rather than the *rcsdiff* program.

As an example, one might set up a Last-Modified field at the bottom of an HTML document to be a link to the *rlog* script, with the document name specified as a parameter. After clicking on this unobtrusive field, the user would be able to see the history of the document.

8.2 Fixed Pages

In addition to permitting individuals to archive URLs of interest to them and find out about modifications to those URLs, AIDE can provide a community of users with specialized “What’s New” pages that report when any of a fixed set of URLs has been changed. Rather than having users specify when to archive a new version, each page is automatically archived as soon as a change is detected. Then users can easily see the most recent changes to a page using *HtmlDiff*, and they can also use the History feature to see earlier versions they may have missed.

Automatic archival of new versions is useful in this context, but it has the disadvantage of increasing disk space dramatically when the entire contents of the page changes (such as the “What’s New in Mosaic” page). When the entire contents are replaced, there is no use for *HtmlDiff*. Automatic archival would still be useful

in cases when one wants a way to go back to arbitrary old versions, but in many cases (including this example), the content provider has its own archive.

8.3 Server-side URL Tracking

Currently, *w3newer* runs on the user's machine, so multiple instantiations of the script may perform the same work. Although it runs a related daemon on the same machine as an AT&T-wide proxy-caching server, which returns information about pages that are currently cached on the server and may eliminate some accesses over the Internet, there is insufficient locality in that cache for it to eliminate a significant fraction of requests.

Alternatively, *w3newer* could be run on the set of pages that have been saved by the *snapshot* daemon. Regardless of how many users have registered an interest in a page, it need only be checked once; if changed, the new version could be saved automatically. Then a user could request a list of all pages that have been saved away, and get an indication of which pages have changed since they were saved by the user.

Adding this functionality would be useful, since it would offer economies of scale. In fact, it could be further extended to be integrated with a "web crawler" and track modifications to pages *pointed to* by pages specified by the user. Following links recursively is inappropriate for tools run by every user individually but would be feasible for a centralized service. It would have the advantage of handling multiple styles of pages, for example:

Virtual Library pages Pages with pointers to many other pages scattered throughout the W^3 could have each link followed to indicate when the referenced pages have been modified, thus eliminating the need for a user to include many pages of interest separately.

Collections of related pages Many times, a "home page" refers to a number of other pages, both within the same namespace and external. By following the internal pages automatically, a single entry in one's hotlist could result in notification whenever any of those pages is modified. *Html-Diff* could in turn be invoked recursively, as described above in Section 5.3.

On the other hand, centralized tracking of modifications would have the disadvantage of being decoupled from a given user's W^3 browser history; i.e., if a user views a page directly, the *snapshot* facility would have no indication of this and might present the page as having been modified. Java might be suitable for conveying that information to the server.

Modifications to support server-side tracking of modifications, including hierarchical tracking, are nearly complete.

8.4 Interactions with CGI scripts

Because AIDE can handle arbitrary URLs, it can interact with CGI scripts that use the GET protocol by passing arguments to the script as part of the URL. However, services that use POST cannot be accessed, because the input to the services is not stored.

Both *w3newer* and *snapshot* would have to be modified to support the POST protocol, in order to invoke a service and see if the result has changed, and then to store away the result and display the changes if it has. The interface to AIDE to support POST is unclear, however. A user could manually save the source to an HTML form and change the URL the form invokes to be something provided by AIDE. It, in turn, would have to make a copy of its input to pass along to the actual service.

Instead, the browser could be modified to have better support for forms:

- It should store the filled-out version of a form in its bookmark file, so the user could jump directly to the output of a CGI script.
- It should be able to pass a form directly to AIDE, along with the URL specified in the FORM tag, so that the output could be stored under RCS.

9 Conclusions

AIDE combines notification, archiving, and differencing of W^3 pages into a single cohesive tool. It achieves economies of scale by avoiding unnecessary HTTP accesses, saving pages at most once each time they are modified (regardless of the number of users who track it), and using RCS as the underlying versioning system. Automatic generation of differences within the HTML framework provides users with the ability to see both insertions and deletions in a convenient fashion.

In the general setting of the W^3 and document retrieval, AIDE benefits two communities: users of the W^3 no longer have to browse to find pages of interest that have changed; HTML providers no longer have to create suitably marked-up pages to show "what's new". While such automation is clearly helpful in this general context, we expect that AIDE will be a critical part of more focused uses of the W^3 , especially in areas involving collaborative and distributed work.

Several issues still need to be addressed. In particular, many of the complications of AIDE could be avoided by better integration with W^3 browsers and

servers. The increasing availability of distributed, hierarchical HTTP repositories such as Harvest [1] will also be both an opportunity and a challenge for scalable notification mechanisms and version archives.

For more information on AIDE, see URL <http://www.research.att.com/orgs/ssr/people/douglis/aide>.

Acknowledgments

Robin Chen, Steve Crandall, John Ellson, P. Krishnan, Mark Rajcok, Herman Rao, and the USENIX referees provided comments on earlier drafts of this paper. Brooks Cutter wrote the version of *w3new* from which the *w3newer* script is derived. Thanks also to David Ladd for numerous discussions about *HtmlDiff*, to Rich Brandwein, Robin Chen, John Ellson, and Herman Rao for discussions about HTTP and HTML; and to the many colleagues who tried out AIDE and provided feedback. Finally, Charles Babbage invented the first computer and called it the “Difference Engine,” a term we appropriated for a new context.

Java is a trademark of Sun Microsystems. Netscape is a trademark of Netscape Communications. UNIX is a registered trademark of X/Open. Windows is a trademark of Microsoft Corporation.

References

- [1] C. Mic Bowman et al. Harvest: A scalable, customizable discovery and access system. Technical Report CU-CS-732-94, Dept. of Computer Science, University of Colorado–Boulder, March 1995.
- [2] K. Church. Char_align: A program for aligning parallel texts at the character level. In *Association for Computational Linguistics*, pages 1–8, 1993.
- [3] Ward Cunningham. WikiWikiWeb. <http://c2.com/cgi-bin/wiki>.
- [4] B. B. Cutter III. w3new. <http://www.stuff.com/~bcutter/programs/w3new/w3new.html>.
- [5] Roome W. D. 3DFS: A time-oriented file server. In *Proceedings of the USENIX 1992 Winter Conference*, pages 405–418, January 1992.
- [6] David Goldberg, David Nichols, Brian M Oki, and Douglas Terry. Using collaborative filtering to weave an information tapestry. *Communications of the ACM*, 35(12):61–70, December 1992.
- [7] James S. Gwertzman and Margo Seltzer. The case for geographical push-caching. In *Proceedings of the Fifth Workshop in Hot Topics in Operating Systems (HOTOS-V)*, pages 51–55, Orcas Island, WA, May 1995. IEEE.
- [8] D. S. Hirschberg. Algorithms for the longest common subsequence problem. *Journal of the ACM*, 24(4):664–675, October 1977.
- [9] J. Howard et al. Scale and performance in a distributed file system. *ACM Transactions on Computer Systems*, 6(1):51–81, February 1988.
- [10] J. W. Hunt and M. D. McIlroy. An algorithm for differential file comparison. Technical Report Computing Science TR #41, Bell Laboratories, Murray Hill, N.J., 1975.
- [11] Java. <http://www.javasoftware.com/>.
- [12] M. Nelson, B. Welch, and J. Ousterhout. Caching in the Sprite network file system. *ACM Transactions on Computer Systems*, 6(1):134–154, February 1988.
- [13] M. Newbery. Katipo. <http://www.vuw.ac.nz/~newbery/Katipo.html>.
- [14] A standard for robot exclusion. <http://web.nexor.co.uk/mak/doc/robots/norobots.html>.
- [15] David S. Rosenblum and Balachander Krishnamurthy. Generalized event-action handling. In Balachander Krishnamurthy, editor, *Practical Reusable UNIX Software*, chapter 9. John Wiley & Sons, New York, 1995.
- [16] Specter, Inc. Webwatch. <http://www.specter.com/users/janos/webwatch/index.html>.
- [17] Sun Microsystems. *The HotJava Browser: A White Paper*. Available as <http://java.sun.com/1.0alpha3/doc/overview/hotjava/browser-whitepaper.ps>.
- [18] W. Tichy. RCS: a system for version control. *Software—Practice & Experience*, 15(7):637–654, July 1985.
- [19] Url-minder. <http://www.netmind.com/URL-minder/URL-minder.html>.
- [20] Warmlist. <http://glimpse.cs.arizona.edu:1994/~paul/warmlist/>.

Author Information

Fred Douglis is a member of technical staff at AT&T Bell Laboratories. His research interests include the *W³*, mobile and distributed computing, and file systems. He received a B.S. from Yale University (1984) and the M.S. (1987) and Ph.D. (1990) degrees from the University of California, Berkeley, all in Computer Science. Email: douglis@research.att.com.

Thomas Ball is a member of technical staff at AT&T Bell Laboratories. His research interests include programming languages, software tools, techniques for efficiently monitoring system and program behavior, and software and system performance visualization. He received a B.A. from Cornell University (1987) and the M.S. (1989) and Ph.D. (1993) degrees from the University of Wisconsin-Madison, all in Computer Science. Email: tball@research.att.com.

Implementation of a Reliable Remote Memory Pager

Evangelos P. Markatos and George Dramitinos*

Computer Architecture and VLSI Systems Group

Institute of Computer Science (ICS)

Foundation for Research & Technology – Hellas (FORTH), Crete

Abstract

Traditional operating systems use magnetic disks as paging devices, even though the cost of a disk transfer measured in processor cycles continues to increase.

In this paper we explore the use of remote main memory for paging. We describe the design, implementation and evaluation of a pager that uses main memory of remote workstations as a faster-than-disk paging device and provides reliability in case of single workstation failures. Our pager has been implemented as a block device driver linked to the DEC OSF/1 operating system, without any modifications to the kernel code. Using several test applications we measure the performance of remote memory paging over an Ethernet interconnection network and find it to be faster than traditional disk paging. We evaluate the performance of various reliability policies and prove their feasibility even over low bandwidth networks, like Ethernet.

We conclude that the benefits of reliable remote memory paging in workstation clusters are significant today and will probably increase in the near future.

1 Introduction

Applications like multimedia, windowing systems, scientific computations, engineering simulations, etc. running on workstation clusters (or networks of PCs) require an ever increasing amount of memory, usually more than any *single* workstation has available. To alleviate the memory shortage problem, an application could use the virtual memory paging provided by the operating system, and have some of its data in main mem-

ory and the rest on the disk. Unfortunately, as the disparity between processor and disk speeds becomes ever increasing, the cost of paging to a magnetic disk becomes unacceptable. Faster swap disks would only temporarily remedy the situation, because processor speeds are improving at a much higher rate than disk speeds [14]. Clearly, if paging is going to have reasonable overhead, a new paging device is needed. This device should provide high bandwidth and low latency. Fortunately, a device with these characteristics exists in most distributed systems and it is not used most of the time. It is the collective memory of all computers in the distributed system, hereafter called *remote memory*.

Remote memory provides high transfer rates which are mainly dictated by the interconnection network. Fortunately, most of the time remote main memory is unused and thus can be exploited by remote memory paging systems. To verify this claim, we profiled the unused memory of the workstations in our lab¹ for the duration of one week: 16 workstations with a total of 800 MBytes of main memory. Figure 1 plots the free memory as a function of the day of the week. We see that for significant periods of time more than 700 Mbytes are unused, especially during the nights, and the weekend. Although during business hours the amount of free memory falls, it is rarely lower than 400 Mbytes!

Architecture and software developments suggest that the use of remote memory for paging purposes is desirable, possible and efficient:

• Memory to memory transfer rates

¹We expect that more main memory will be available in places that have lighter load. Our workstations are heavily used running VERILOG simulations for most of the time.

*The authors are also with the University of Crete.

between workstations have increased sharply in the last few years: Local Area Networks (like ATM and FDDI) have a high throughput and (usually) low latency. This increase in communication bandwidth implies a dramatic decrease in network transfer time for large messages (like operating system pages). On the other hand, the disk technology has *not* shown a similar increase in transfer rates. Moreover, disk accesses suffer from seek and rotation latency which is not expected to be reduced from advances in semiconductor technology.

- **Application's working sets have increased dramatically over the last few years:** Modern processors provide 64-bit address spaces, which make it possible for the processor to address an enormous amount of memory. Thus, software that takes advantage of a large address space is being developed: memory-mapped files and databases, sophisticated window interfaces, and multimedia, are a few examples that require an enormous amount of main memory.
- **Modern architectures provide low latency remote memory accesses:** Modern distributed systems provide a variety of efficient access operations to remote memories. The SCI-to-SBUS interface provides SPARC workstations with the ability to access the memories of other workstations in a network using simple load and store operations [23]. Similar ability is provided by Telegraphos [19], Hamlyn [5], Memory Channel [13], and SHRIMP [4]. Fast remote memory accesses have also been implemented in software using Active Messages [26, 2], programmed network interfaces [16], and trap-based remote invocation [25]. The ability to perform single remote memory accesses efficiently will enhance the performance of a remote memory paging policy, since the application can use them to access infrequently used pages.

In this paper we show that it is both possible and beneficial to use remote memory as a reliable paging device by building the systems software that transparently transfers operating system pages across workstation memories within a workstation cluster. We describe a pager built as a device driver of the DEC OSF/1 operating system. Our pager is completely portable to any system

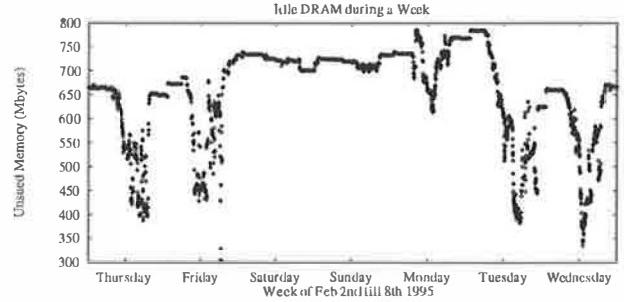


Figure 1: **Unused memory in a workstation cluster.** The figure plots the idle memory during a typical week in the workstations of our lab: a total of 16 workstations with about 800 Mbytes of total memory. We see that memory usage was at each peak (and thus free memory was scarce) at noon and afternoon of working days. In all times though, more than 300 Mbytes of main memory were unused.

that runs DEC OSF/1, because we didn't modify the operating system kernel. More important, by running real applications on top of our memory manager, we show that even on top of low bandwidth interconnection networks (like Ethernet), it is *efficient* to use remote memory as backing store. Our performance results suggest that paging to remote memory over Ethernet, rather than paging to a local disk of comparable bandwidth, results in up to 96% faster execution times for real applications. Moreover, we show that reliability and redundancy comes at no significant extra cost. We describe the implementation and evaluation of several reliability policies that keep some form of redundant information, which enables the application to recover its data in case a workstation in the distributed system crashes. Finally, we use extrapolation to find the performance of paging to remote memory over faster than Ethernet networks like FDDI and ATM. Our extrapolated results suggest that paging over a 100 Mbits/sec interconnection network, reduces paging overhead to less than 17% of the execution time of the application running over such a network. Faster networks will reduce this overhead even more.

The rest of the paper is organized as follows: Section 2 presents the design of a remote memory pager and the issues involved. Section 3 presents the implementation of the pager as a device driver. Section 4 presents our performance results which are very encouraging. Section 5 presents some aspects that we plan to explore as part of our future work. Section 6 presents related work. Finally,

section 7 presents our conclusions.

2 The Design of a Remote Memory Pager

2.1 Selection of Workstations

All workstations, that participate in remote memory paging are registered in a common file. These workstations are known as remote memory servers, while the workstations that run applications that use remote memory for swapping are called clients. Depending on its workload, a workstation may act either as a server, or as a client.

All server workstations run a remote memory server that handles requests for pageins, pageouts, as well as for swap space allocation. When a client wants to swap out memory it picks the most promising server, asks for a number of page frames and starts sending requests to it. When a server runs out of memory, it denies further swap space allocation requests. When native memory-demanding processes start on a server workstation, part of the server's memory is swapped out to disk. Future requests will be serviced from the disk, and a note will be sent to the client, advising it to send no more pages to this server. On reception of this message, the client will try to find another server having enough free memory and migrate the pages that were stored by the loaded server to the new one. If no server having enough free memory can be found the client's local disk will be used to house these pages. Whenever the client's local disk is used to store some of its paged out pages, the client periodically checks the memory load of all possible remote memory servers. If a server having enough free memory is found, some of the pages stored at the local disk are replicated to this server. Future requests concerning these pages will be served by the remote memory server rather than the disk.

2.2 Reliability

In a distributed system, a workstation may crash at any time. If the crashed workstation acts as a server, it will lose the pages of several clients. Clearly, it is not acceptable for applications running on the client workstation to crash due to remote server crash. Instead, we would like to be able to recover their pages. Otherwise a remote server crash will cause a client crash as well, since all programs that have some of their pages swapped out (including programs like init and system daemons) will not be able to continue execution.

There are many types of crashes. First of all there may be machine crashes due to a black out. This situation is not addressed by this paper, since most computer buildings are equipped with UPSs. Another cause of failure may be a network problem (e.g. network partitioning due to a bridge failure). In this case, the client can not retrieve its pages from the servers. As a result it remains blocked waiting for the network to recover. The most frequent cause of crash is a software crash, followed by a hardware error. To avoid loss of data due to a server crash, some systems write all network memory pages to the disk as well ([1, 11]). Instead we implement a *reliable* remote memory paging system that is able to reconstruct the lost pages.

To provide this level of reliability, some form of redundancy must be used. The main issues that must be taken into account regarding the form of redundancy used are:

- The runtime overhead introduced must be minimal since it is a cost paid even when no server crashes.
- The memory overhead introduced must be as low as possible because the memory reserved for reliability could be used in order to store memory pages of other workstations.
- The crash recovery overhead, that is the time it takes to recover from a server crash. This overhead is not as important as the previous two, since it is affordable to devote a few more seconds whenever a server crashes, which happens rather rarely.

We explore three different policies: mirroring, parity, and parity logging.

Mirroring: The simplest form of redundancy is *mirroring*. In mirroring, there exist two copies of each page. When the client swaps out a page, the page is sent to two different servers. Even when one of the servers crashes, the application is able to complete its execution, because all pages of the crashed server exist on the mirror servers. Obviously the crash recovery overhead, in case of mirroring, is minimal. However, the runtime overhead is rather high, since each pageout requires two page transfers. To make matters worse, mirroring wastes half of the remote memory used.

Parity: To reduce the main memory waste caused by mirroring, we can use parity-based re-

dundancy schemes much like the ones used in RAIDS [6]. Suppose, for example, that we have S servers, each having P pages. Page (i, j) is the j_{th} page that resides on server i . Assume, that we have P parity pages, where parity page j is formed by taking the XOR of all the j_{th} pages in all servers. We say that all these j_{th} pages belong to the same parity group. If a server crashes, all its pages can be restored by XORing all pages within each parity group.

When the client swaps out a page it has to update the parity to reflect the change. This update is done in two steps:

1. The client sends the swapped out page to the server, which computes the XOR of the old and the new page.
2. The server sends the just computed XOR to the parity server, which XORs it with the old parity, forming the new parity.

Unfortunately, this method involves two page transfers: one from client to server, and one from server to parity. Moreover, the client should not discard the page just swapped out, because the server may crash before the new parity is computed, thus, making it impossible to restore the swapped out page. This parity method increases the amount of remote main memory only by a factor of $(1 + 1/S)$ minimizing the memory overhead, but it still imposes a significant runtime overhead.

Parity Logging: To avoid the additional page transfers induced by the basic parity method, we have developed a parity logging scheme. The key idea is that a given page need not be bound to a particular server or parity group. Instead, every time a page is paged out, a new server and a new parity group may be used to host the page.

Suppose the client uses S servers. Each paged out page is XORed with a page size buffer maintained by the client (which is initially filled with zeros) and then is transferred to a server following a round robin policy. Whenever S pages have been transferred, the buffer is also transferred to a parity server. Using this technique, the runtime overhead is minimal, since for each paged out page $1 + 1/S$ page transfers are required. When a server crashes, all of its pages can be restored by XORing the pages in their group with the corresponding parity page.²

²Note that since the parity page is computed by the

Every time a page is repaged out, it is marked in the old parity group containing it as inactive.³ When all the pages of a parity group are marked as inactive, all the memory server pages and the corresponding parity page can be reused. It is obvious that each memory server must have some extra overflow memory to support parity logging since many versions of a given page may be present simultaneously at the servers' memory. Also, due to this situation, it is possible that some server runs out of memory. In this case, one has to perform garbage collection freeing parity sets by combining their active pages to new ones. In our experiments, 4 servers were used devoting 10% more memory to support parity logging and we never had to perform garbage collection.

3 Implementation

The proposed system has been built and is in everyday use. It consists of a client issuing paging requests and servers satisfying these requests. It is also able to use the local disk for paging and may support either mirroring or parity logging. The client side has been linked with the DEC OSF/1 kernel of a DEC-Alpha 3000 model 300 with 32 MB main memory as a block device driver that handles all pagein and pageout requests. In order to service these requests, it may forward them either to user level servers running on other hosts, or to the local disk. The DEC OSF/1 kernel is not even aware that we use remote main memory instead of magnetic disk as a paging device. It just performs ordinary paging activities using a block device. This design minimizes the modifications needed in order to port the system to another operating system and avoids modifications to the operating system kernel.

3.1 The Remote Memory Pager

Normally the Remote Memory Pager (RMP for short) is a client which forwards the paging requests to a remote server using sockets over an Ethernet. The RMP connects to the remote memory servers using sockets over TCP/IP. One dedicated paging daemon issues pagein and pageout requests to the server and receives the data sent by them. When mirroring is used, it is responsible for selecting two servers for each paged out page

client, it is not necessary to wait for acknowledgments from the servers before transferring the parity page in order to be able to recover from a single server crash.

³However, the old version of the page is *not* deleted from the server's memory, because if it were, the *old* parity page should be updated, leading to more page transfers.

and transfer the data to them. When parity logging is used, it maintains all the data structures related to page and parity group management and computes the parity pages. Security is ensured by allowing access to our device only to the superuser and by using privileged ports for the communication among the client and the servers.

RMP is also capable of forwarding the requests to the local disk using either a specified partition or a file. In the former case, it invokes a routine that places the request in the disk queue. In the later case it issues a read or write operation through the VFS layer routines. When no server can be found in order to satisfy the client's requests, paging to local disk is used.

Although the current implementation runs on top of a low bandwidth 10 Mbps Ethernet, remote paging is up to 2 times faster than using a local disk of the same bandwidth. It takes about 8.4 ms to transfer an 8KB page through the network, while transferring a page to/from the local disk takes about 17 ms. Faster networks such as ATM, or FDDI should offer even more promising performance, especially when faster communication protocols are used [26].

3.2 The Remote Memory Server

The server is a user level program listening to a socket and accepting connections from clients. Each client is served by a new instance of the server which uses portion of the local workstation's main memory to store the client's pages. When the client requests a pagein, the server transfers the requested page(s) over the socket. When the client requests a pageout, the server reads the incoming pages from the socket, and stores them in its main memory. The server is also responsible for swap space allocation and for providing periodically information to the client concerning the memory load of its host. A parity server is by no means different than a memory server. It just performs pageins and pageouts responding to client requests without knowing whether it stores memory pages or parity pages.

4 Performance Results

To evaluate the performance of our remote memory pager, and compare it to traditional disk paging, we conducted a series of performance measurements using a number of representative applications that require a large amount of memory. Our applications include **GAUSS**, a gaus-

sian elimination, **QSORT**, a quicksort program, **FFT**, a Fast-Fourier Transform, **MVEC**, a matrix-vector multiplication, **FILTER**, a two pass separable image sharpening filter described in [20] and **CC**, a kernel build after modifying the code of our device driver. All applications were executed on the DEC-Alpha 3000 model 300, and were compiled with the standard C compiler with the optimization enabled. All workstations that contributed their main memory for paging purposes were DEC-Alpha 3000 model 300, connected via a standard 10Mbits/sec Ethernet. In all experiments the amount of idle memory was larger than the amount of memory needed for paging and was equally distributed among all workstations. The local disk that was used for paging is a DEC RZ55, providing 10Mbits/sec bandwidth, and average seek time of 16 msec.

4.1 Performance of Remote Memory Paging Over the Ethernet

In our first experiment we evaluate four methods for paging:

- **NO_RELIABILITY**, which uses only main memory of other workstations as a paging device. In this experiment two remote memory servers were used. The measurements were done on an (almost) idle Ethernet to ensure repeatability.
- **PARITY_LOGGING**, which uses 4 servers plus a parity server, all devoting 10% overflow memory.
- **MIRRORING**, which uses one primary memory server and one mirror memory server.
- **DISK**, which uses the local DEC RZ55 disk for paging. In this case the page transfer requests go directly from the DEC OSF/1 kernel to the disk driver without the intervention of our pager.

The completion time of the applications is plotted in figure 2. We see that in all cases the use of remote memory results in significant performance improvements. For example, for the **GAUSS** application, the **NO_RELIABILITY** results in 96% faster execution time than **DISK**. Even for the **MVEC** application which performed very little paging, **NO_RELIABILITY** results in 32% faster execution time. The reliability methods induce some runtime overhead as expected but still perform

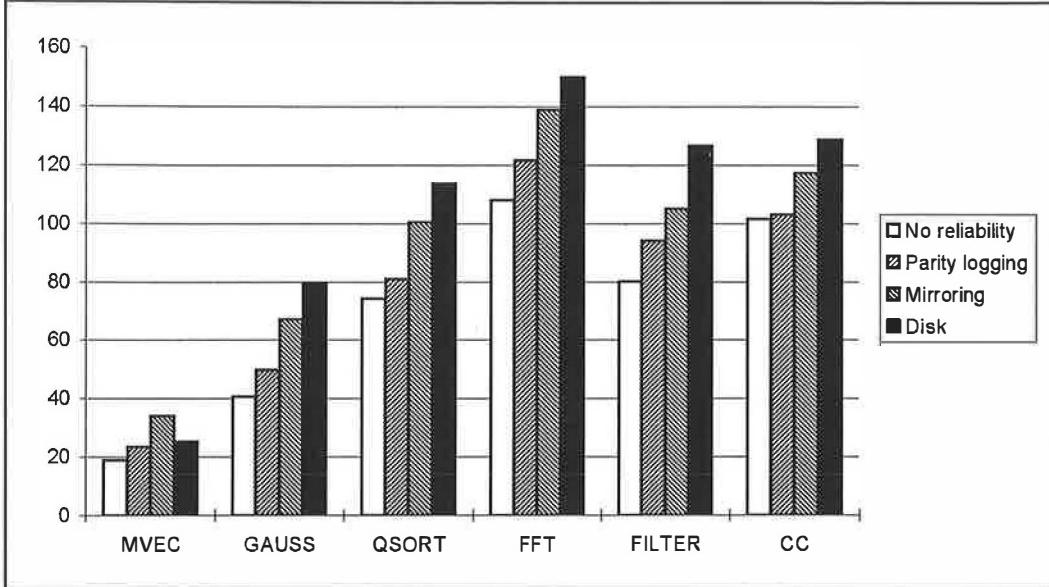


Figure 2: **Performance of applications using either the disk, or the remote memory as paging device.** We see that for all applications, the use of remote memory results in significantly faster execution. All applications were run on a DEC-Alpha 3000 model 300 workstation. The input sizes for QSORT was 3000 records, for GAUSS, a 1700×1700 matrix, for MVEC, a 2100×2100 matrix, for FFT an array with 700 K elements, for FILTER a 12 MB image, and the whole DEC OSF/1 V3.2 kernel for CC.

much better than DISK. PARITY_LOGGING results in 40.4% faster execution time for QSORT and in 59.86% faster time for GAUSS. MIRRORING also performs better than DISK for all applications except MVEC, since MVEC performs many pageouts and almost no pageins.

In order to evaluate the use of remote memory for a more realistic application, we measured the completion time of a kernel build after modifying the code of our device driver. As can be seen in figure 2, NO_RELIABILITY performs 26.56% better than disk, PARITY_LOGGING performs 24.65% better and MIRRORING performs just 9.7% better. We see that PARITY_LOGGING performs very close to NO_RELIABILITY. As the number of the remote memory servers used increases, the difference in performance between NO_RELIABILITY and PARITY_LOGGING becomes lower.

Our performance results suggest that paging to remote memory over an Ethernet interconnection network is simply faster than paging to the disk. Even though both the disk and the Ethernet have similar data transfer rates, remote memory does not suffer from seek and rotational latency as DISK does.

Our experimental results verify that even when the network data transfer rate is as low as the disk transfer rate, the performance of remote memory is significantly higher than the performance of disk. Moreover the performance requirements of reliability are surprisingly small. Since architecture trends suggest that modern high speed networks provide much higher data transfer rates than modern disks, the performance improvements of remote memory over disk are bound to increase.

4.2 Scaling the Input

To understand the impact of the working set size on the paging policy, we measure the execution time of one of our applications (FFT), as a function of its input size. The completion time of FFT both under PARITY_LOGGING and under DISK is plotted in figure 3. We see that as soon as the working set size exceeds 18 MBytes, the paging starts, and the completion time of the application rises sharply. Most users would not be willing to tolerate such a high overhead in order to run an application that does not fit in main memory. Fortunately, remote memory reduces this overhead substantially.

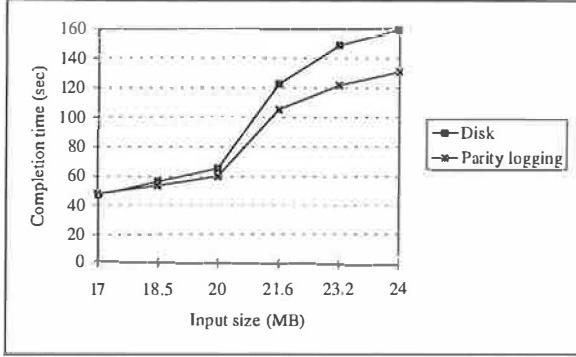


Figure 3: Performance of FFT as a function of input size when either the disk, or remote memory are used as backing store.

4.3 Scaling the Network Bandwidth

Although figure 3 suggests that the performance of remote memory (parity logging) is significantly better than the performance of disk, the completion time of an application even under remote memory may be unacceptably high. Hopefully, the performance of remote memory will be improved as soon as the Ethernet interconnection network is substituted by a faster one (e.g. FDDI, ATM, FCS, etc.). To evaluate the performance of the applications on top of faster networks we make detailed performance measurements that separate the completion time of the application into the following factors: (i) user time (*utime*), (ii) system time (*systime*) (iii) initialization time (*inittime*) (iv) page transfer time (*ptime*). Using the provided `time` command we measure the *utime*, *systime*, and elapsed time (*etime*) for each application. Subtracting the *utime* and *systime* from the *etime* for instances of the applications that perform no paging we calculate the *inittime*, that is the time it takes the operating system to load and start executing the application. The *ptime* consists of the protocol processing time (*pptime*) and the bandwidth dependent blocking time (*btime*). We measured the *pptime* and found it to be equal to 1.6 ms per page for TCP/IP. We calculate the *btime* using the formula : $btime = (etime - utime - systime - inittime - \text{no_of_page_transfers} * pptime)$. Assuming that a network with X times higher bandwidth will decrease *btime* by a factor of X , we can predict the *etime* of the application over this high bandwidth network. Thus, the formula used is : $\text{Expected_elapsed_time} = utime + systime + inittime + \text{number_of_page_transfers} * pptime + btime/X$.

$btime/X$.

We made all these measurements on our FFT application, and predict its performance on a system with an interconnection network which provides ten times more bandwidth than the Ethernet. We also predict its completion time on a system that has enough memory to hold all the working set of the application (**ALL_MEMORY**) by adding the *utime*, *systime* and *inittime*. The predicted execution times, along with the measured execution times of **DISK** and **PARITY_LOGGING** are plotted in figure 4. We see that **ETHERNET*10** performs very close to **ALL_MEMORY**, and significantly better than both **ETHERNET** and **DISK**.

To understand the results shown in figure 4, we analyze the execution time of FFT with 24MBytes of input when **PARITY_LOGGING** is used. The measured elapsed time is 130.76 seconds, consisting of 66.138 sec of useful user time, 3.133 sec of system time, 0.21 sec of initialization time and 61.279 sec of page transfer time. During the same run, the application suffered 2718 pageouts and 2055 pageins. Since 4 servers were used plus a parity server the number of page transfers was equal to $3397 + 2055 = 5452$. Thus the protocol overhead was equal to $5452 * 0.0016$, or about 8.723 sec. The bandwidth dependent blocking time was equal to $61.279 - 8.723$, or about 52.556 sec. Using a ten times faster interconnection network, the bandwidth dependent waiting time will be reduced to 5.255 sec. Thus, the total completion time of FFT would be $66.138 + 3.133 + 0.21 + 8.723 + 5.255$ sec, or 83.459 sec, divided as follows: 79.246% in user time, 3.754% in system time, 0.252% in initialization time and 16.748% in page transfer time. We see that a 100 Mbit/sec interconnection network reduces the total paging overhead to less than 17% of the total application execution time. We believe that most users would be willing to pay such an overhead in order to run an application that does not fit in main memory. After all, the only other option they have is to suffer from disk thrashing.

4.4 The Latency of Remote Memory Paging

As explained previously, the paging latency for FFT with input size equal to 24 MB is 61.279 sec, or 11.24 ms per page transfer. From these, 1.6 ms were spent during protocol processing and 9.64 ms were spent transferring each page on the Ethernet.

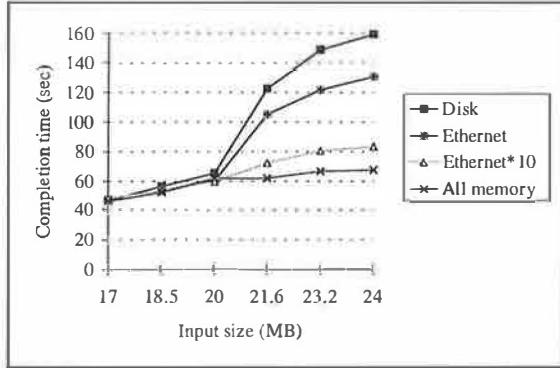


Figure 4: **Performance of FFT for various Architecture Alternatives.** DISK is the measured completion time when paging to a local disk. ETHERNET is the measured completion time of parity logging to remote memory on top of the Ethernet. ETHERNET*10 is the predicted completion time when using remote memory as a paging device, on top of a network that provides ten times more bandwidth than the Ethernet interconnection network. ALL.MEMORY is the predicted completion time of FFT when we use the same workstation but with enough memory to hold its entire working set.

Previous measurements have reported that a 4 KByte page takes about 45 ms over an Ethernet for each page in [22]. Of those 45 ms, 19 ms were spent on TCP overhead, 4 ms were spent on Mach IPC overhead, 7.2 ms were spent on the Ethernet, and the rest were spent on the computer's I/O bus. The total software latency of our implementation, is only 1.6 ms. The reason for this significant difference in performance is threefold:

- The I/O bus of the DEC-Alpha 3000 model 300 we use is significantly faster and does not pose a problem in performance.
- We use a DEC-Alpha processor, which is 3-4 times faster than the 386 processor used in [22].
- Finally, our pager is implemented as a block device driver, while in [22] it was implemented as a user-level memory manager on top of Mach. Although user-level memory management gives increasing flexibility it induces large overhead.

In general, although our approach may have less flexibility than a full-fledged user-level pager,

it has much better performance. Moreover, our device-driver implementation provides better performance than traditional (local) disk paging, while user-level implementations have not reported performance results to support similar claims [22].

4.5 Using Busy Workstations as Servers

In all our experiments so far, the remote memory servers run on idle workstations. However, workstations that are able to donate their memory for paging purposes may not be completely idle, as they may run interactive applications. Thus, we would like to investigate how our performance figures change when a non-idle workstation is used as a memory server. So, we conducted the following experiment:

On each server workstation we started an X-window environment, and an instance of the vi editor which was continuously used for editing. Then, we run the applications of the experiment in figure 2. The same inputs, and the same clients were used. The only difference was that the remote memory server processes were run on busy instead of idle workstations.⁴

We were surprised to see that for the FFT, GAUSS, and MVEC applications, their completion times were within 1 sec of their completion times when the server ran on an idle workstation. Only QSORT suffered a 7% overhead in its completion time: probably the kernel swapped out some of the remote memory server's pages on the disk. However, in order to find out how the completion time of our applications changes with server load, we ran FFT and QSORT under NO.RELIABILITY using two remote memory servers. On one of them a cpu bound program (performing a "while(1);") loop) was initiated. To our surprise, even then the completion times of our applications were within 7% of their completion times when the server ran on an idle workstation.

Our performance figures suggest that most of the time the remote memory servers were able to satisfy the client's requests immediately, even on busy workstations. Our results agree with the

⁴ One could argue that an X-window environment and an editor, induce almost no load on the workstation. But, this is exactly the point: a typical workstation, even when it is used, it is very lightly loaded. The rest of the workstations that are heavily loaded do not donate their main memory for remote paging.

measurements in figure 1 which report that a significant portion of all workstation's memory is unused even at business hours, thus no overhead is expected to be seen when some other server process uses the extra pages.

In the same course of experiments, we would like to see what is the overhead that remote paging induces on the server workstation. Thus, we measured the CPU utilization of the (otherwise idle) remote memory server for all our experiments, and found it always to be less than 15%. Thus, the computational overhead imposed on the remote workstation is so low that will not be noticed by the workstation's owner.

4.6 Using Remote Memory Paging over a Loaded Ethernet

All the experiments presented so far were done over an almost idle Ethernet to ensure repeatability of our results. However, we would like to find out how the performance of remote memory paging is affected by the load of the network. That is why we repeated our experiments using an already loaded Ethernet. The results showed a performance degradation even when the Ethernet was lightly loaded. This situation is by no means surprising since the paging itself uses all the bandwidth it can get. Adding more sources of traffic leads to an enormous demand for bandwidth causing repeated collisions and lowering the effective bandwidth of the network, leading to throughput collapse.

Fortunately, this inefficiency is not inherent to remote memory paging but rather to the CSMA/CD protocol employed by the Ethernet [24]. This means that it is still beneficial to use remote memory paging over networks that employ other technologies (e.g. token ring), as long as they are able to provide to remote memory paging an effective bandwidth of 10 or more Mbps.

4.7 Using the Local Disk to Increase Reliability

In our work we use remote main memory to store redundant information that will be used to recover from workstation crashes. Another approach would be to store all remote pages to the local disk as well [11], effectively treating remote memory as a write-through cache of the disk. We will now compare the two approaches to find out the circumstances under which the one approach is preferable to the other.

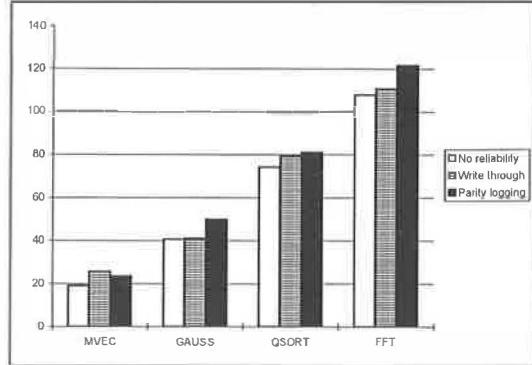


Figure 5: Performance of parity logging and write through for various applications. The input sizes for QSORT was 3000 records, for GAUSS, a 1700×1700 matrix, for MVEC a 2100×2100 matrix, and for FFT an array with 700 K elements.

Both approaches use the remote memory to satisfy the read requests. This means that both approaches perform reads at the same speed and avoid disk head movements due to reads, thus outperforming the local disk. *Parity logging* transfers $1 + 1/N$ pages per paged out page, due to the parity computation (in our experiments N was equal to 4). On the other hand, *write through* transfers each paged out page both to disk to the remote memory. These two page transfers are executed in parallel. This means that the choice of the right approach depends on the effective bandwidth offered by the disk and the network. If the network bandwidth is much higher than the disk bandwidth, then the disk will be the bottleneck for *write through* making it an unwise choice. If however the effective bandwidth offered by the disk is comparable to the bandwidth offered by the network and the system can overlap disk transfers with network transfers then it is unclear which method is best to use. In our experimental environment the disk and network bandwidth are both equal to 10 Mbps. When *write through* is used the effective disk bandwidth is close to 10 Mbps, since there are no head movements for reads and writes are performed in large chunks. In this environment *write through* performs better than *parity logging* and slightly worse than our no-reliability implementation in most cases, as shown in figure 5. However, when a modern high bandwidth network is used, *parity logging* will probably be the best approach, since *write through* will eventually be limited by the local disk bandwidth.

5 Discussion - Future work

Our implementation suggests that it is possible to build a reliable *efficient* remote memory pager without making any modifications to the operating system kernel. Although our system contains all necessary mechanisms to support remote memory paging, there are a few more issues concerning the overall *policy* that deserve further investigation. Some of these issues are discussed below.

Network load: Although remote paging is faster than using the local disk, sometimes the network traffic may be so high that the bandwidth used by RMP will be limited. In this case the cost of using the network, especially in the case of old low bandwidth networks like Ethernet, may become higher than the cost of using the local disk. Such a situation could be handled by the RMP by measuring the time it takes to satisfy a request and using a threshold to determine whether it should continue to use the network to route pageout requests or it would be better to switch to the local disk.

Heterogeneous networks: The current implementation assumes a network of workstations that all have the same order of magnitude of physical memory and are interconnected by a local area network. It would be interesting to explore the requirements that heterogeneous networks pose to the design of the remote pager. For example, on a wider area network the time it takes to transfer a page may not be identical for each server. In this case there may be more than three levels in the memory hierarchy (local memory, remote memory, disk), depending on the variance of the cost of communication among the hosts of the network. Connecting machines that have an enormous amount of memory (e.g. a supercomputer) to a network of workstations also poses some problems. When the supercomputer memory is idle, it may not always be easy to find enough free remote workstation memory in order to be able to use reliability policies. In this case, a no reliability policy can be used, since all remote memory will be provided by a single host (the supercomputer).

6 Related Work

Several research groups have studied the issues in using remote memory in a workstation cluster to improve paging performance [2, 12, 7, 15, 22, 3].

Felten and Zahorjan [12] have implemented a remote paging system on top of a traditional Ethernet based system, and presented an analytical model to predict its performance. Their performance results, although preliminary, are encouraging towards using remote memory paging systems. Schilit and Duchamp [22] have implemented a remote memory paging system on top of Mach 2.5 for portable computers. Their remote memory paging system has performance similar to local disk paging. The cost of a single remote memory page in over an Ethernet, they quote, is about 45 ms for a 4Kbyte page, which is rather high. According to their measurements, a significant percentage of this time (close to 16 ms) is spent executing Mach IPC and TCP code. Comer and Griffoen [7] have implemented and compared remote memory paging vs. remote disk paging, over NFS, on an environment with diskless workstations. Their results suggest that remote memory paging can be 20% to 100% faster than remote disk paging, depending on the disk access pattern. Anderson *et. al.* have proposed the use of network memory as backing store [2]. Their simulation results suggest that using remote memory over a 155Mbits/s ATM network "is 5 to 10 times faster than thrashing to disk" [2]. In their subsequent work [18], they outline the implementation of a remote memory pager on top of an ATM based network.

Our work differs from previous approaches to remote memory paging in the following aspects: (i) we use a variety of real applications to evaluate and demonstrate the feasibility of remote memory paging, and (ii) we explore the issues in building a *reliable* remote memory system that is resilient to individual workstation failures. Previous approaches either ignore workstation failures, or write dirty pages both to the disk and the remote memory, limiting their performance by the available disk throughput.

Recently, research groups start to explore the issue of using remote memory to improve file system performance [11, 1, 8]. Feeley *et. al.* have implemented a global memory management system in a workstation cluster, using the idle memory in the cluster to store clean pages of memory loaded workstations [11]. Anderson *et. al.* have implemented xFS, a serverless network file system [1, 9]. Both network memory systems have been incorporated inside the kernel of existing operating systems and their performance has been demonstrated. Although improvements in

file system performance may ultimately lead to paging performance improvements, solutions developed for file systems may be cumbersome, or too general for remote memory paging systems: (i) in file systems, client processes may share file data, leading to *cooperative* remote memory management policies. In paging instead, clients *never* share their swap spaces. Thus, policies developed to optimize a client-server approach to file I/O, and facilitate cooperation among client processes that share data, do not necessarily apply to a paging system where no single paging server is used, and no sharing (of swap spaces) between client processes takes place. Finally, we use the network memory for storing both clean and dirty pages using our novel parity-based approach. Thus, page out (write) operations can be acknowledged at the speed of remote memory, while in [11, 1] page out operations are acknowledged at the speed of disk.

Although the area of reliability in network memory systems is new, it shares several of the ideas developed for other areas of reliable memory management. For example, parity based methods have been extensively used for Redundant Arrays of Inexpensive Disks (RAIDs) [6].

Log based methods have been used for Log based file systems, that send all updates to a file to be written in sequential blocks of the disk [21]. Thus, the head of the disk does not make random seek movements, and the effective data transfer rate of the disk increases. Log based file systems, alike our LOGGING methods, create a fragmented space that needs to be cleaned. Although the general ideas may be similar, there are substantial differences between a log based file system and the log based reliable network memory we propose. For example, (i) Fragmentation in log based file systems occurs in large chunks (several Mbytes), while fragmentation in log based reliable network memory occurs in small parity groups, and (ii) Log based reliable network memory systems may use parity groups as soon as they are emptied, but log based file systems may not use emptied disk blocks, because this would require a head movement. (iii) Cleaning in log based file system is much more infrequent than it is in network memory, thus it must be made more efficient, and (iv) the objective of log based network memory systems is to reduce page transfers, while the objective of log based file systems is to reduce disk head movements. For the above reasons, methods developed for log based file systems do not necessarily apply “as is” to network memory systems.

Our work bears some similarity with distributed shared memory systems [17, 10] in that both approaches use remote memory to store an application’s data. Our main difference is that we focus on *sequential* applications where pages are not (or rarely) shared, while distributed-shared-memory projects deal with parallel applications, where the main focus is to reduce the cost of page sharing.

7 Conclusions

In this paper we explore the use of remote main memory for paging. We describe our prototype implementation of a remote memory pager implemented on top of the DEC OSF/1 operating system as a device driver. No modifications were made to the kernel of the (monolithic) DEC OSF/1 operating system. We run several applications that use our pager on top of a DEC-Alpha-based workstation cluster to measure the performance of the system. The contributions of this paper are:

- We describe how to build a reliable remote memory paging system; we propose a novel parity-based policy that is resilient to single workstation failures.
- We show that reliable paging to remote memory results in substantial performance improvements over local disk paging.

Based on our implementation and our performance results we conclude:

- *Paging to remote memory results in significant performance improvement over paging to disk.* Applications that use our pager even when running on top of *traditional* Ethernet technology show performance improvements of up to 96% (see figure 2). Extrapolating from our results, we show that on top of a faster interconnection network even higher performance improvements are realizable!
- *Paging to remote memory is an inexpensive way to let applications use more main memory than a single workstation provides.* Remote memory paging provides good performance with almost no extra hardware support. The only way for magnetic disks to provide comparable performance is to use expensive disk arrays.

- *Reliability in remote memory paging comes at low cost.* Parity logging based paging provides reliability at low runtime and memory overhead, performs very close to NO.RELIABILITY and much faster than disk paging.
- *The benefits of paging to remote memory will only increase with time.* Current architecture trends suggest that the gap between processor and disk speed continues to widen. Disks are not expected to provide the bandwidth needed by paging unless a breakthrough in disk technology occurs. On the other hand, interconnection network bandwidth keeps increasing at a much higher rate than (single) disk bandwidth, thereby increasing the performance benefits of paging to remote memory.

Based on our performance measurements we believe that remote memory paging is a cost-effective and performance-effective way to execute memory-limited applications on a network of workstations.

Acknowledgments

This work was developed in the ESPRIT/HPCN project “SHIPS”, and will form a test application for the ACTS project “ASICOM”, funded by the European Union (DG III and DG XIII). We deeply appreciate this financial support, without which this work would have not existed.

We would like to thank Manolis Katevenis, Sotiris Ioannidis and Kosmas Papachristos for useful comments in earlier drafts of this paper. Michael J. Feeley and Henry M. Levy pointed out useful references. Finally, we deeply appreciate the thoughtful comments of the anonymous referees.

References

- [1] T. E. Anderson, M. D. Dahlin, J. M. Neefe, D. A. Patterson, D. S. Roselli, and R. Y. Wang. Serverless Network File Systems. In *Proc. 15-th Symposium on Operating Systems Principles*, December 1995.
- [2] Thomas E. Anderson, David E. Culler, and David A. Patterson. A Case for NOW (Networks of Workstations). *IEEE Micro*, February 1995.
- [3] G. Bernard and S. Hamma. Remote Memory Paging in Networks of Workstations. In *Proceedings of the SUUG International Conference on Open Systems: Solutions for Open Word*, April 1994.
- [4] M. Blumrich, K. Li, R. Alpert, C. Dubnicki, E. Felten, and J. Sandberg. Virtual Memory Mapped Network Interface for the SHRIMP Multicomputer. In *Proceedings of the Twenty-First Int. Symposium on Computer Architecture*, pages 142–153, Chicago, IL, April 1994.
- [5] Greg Buzzard, David Jacobson, Scott Marovich, and John Wilkes. Hamlyn: a high-performance network interface with sender-based memory management. In *Proceedings of the Hot Interconnects III Symposium*, August 1995.
- [6] Peter M. Chen, Edward K. Lee, Garth A. Gibson, Randy H. Katz, and David A. Patterson. RAID: High-Performance, Reliable Secondary Storage. *ACM Computing Surveys*, 26(2):145–185, June 1994.
- [7] D. Comer and J. Griffoen. A new design for Distributed Systems: the Remote Memory Model. In *Proceedings of the USENIX Summer Conference*, pages 127–135, 1990.
- [8] T. Cortes, S. Girona, and J. Labarta. PACA: A Distributed File System Cache for Parallel MAchines. Performance under Unix-like workload. Technical Report UPC-DAC-1995-20, Departament d’Arquitectura de computadors, Universitat Politècnica de Catalunya (UPC), June 15 1995.
- [9] M.D. Dahlin, R.Y. Wang, T.E. Anderson, and D.A. Patterson. Cooperative Caching: Using Remote Client Memory to Improve File System Performance. In *First USENIX Symposium on Operating System Design and Implementation*, pages 267–280, 1994.
- [10] G. Delp. *The Architecture and implementation of Memnet: A High-Speed Shared Memory Computer Communication Network*. PhD thesis, University of Delaware, 1988.
- [11] M. J. Feeley, W. E. Morgan, F. H. Pighin, A. R. Karlin, H. M. Levy, and C. A. Thekkath. Implementing Global Memory Management in a Workstation Cluster. In *Proc. 15-th Symposium on Operating Systems Principles*, December 1995.

- [12] E. W. Felten and J. Zahorjan. Issues in the Implementation of a Remote Memory Paging System. Technical Report TR 91-03-09, University of Washington, November 1991.
- [13] R. Gillet. Memory Channel. In *Proceedings of the Hot Interconnects III Symposium*, August 1995.
- [14] J. L. Hennessy and D. A. Patterson. *Computer Architecture: A Quantitative Approach*. Morgan Kaufmann Publishers, Inc., 1990.
- [15] L. Iftode, K. Li, and K. Petersen. Memory Servers for Multicomputers. In *Proceedings of COMPCON 93*, 1993.
- [16] J. Kuskin, D. Ofelt, M. Heinrich, J. Heinelein, R. Simoni, K. Gharachorloo, J. Chapin, D. Nakahira, J. Baxter, M. Horowitz, A. Gupta, M. Rosenblum, and J. Hennessy. The FLASH Multiprocessor. In *Proc. 21-th International Symposium on Comp. Arch.*, pages 302–313, Chicago, IL, April 1994.
- [17] Kai Li and Paul Hudak. Memory Coherence in Shared Virtual Memory Systems. *ACM Transactions on Computer Systems*, 7(4):321–359, November 1989.
- [18] A. Mainwaring, C. Yoshikawa, and K. Wright. NOW White Paper: Network RAM Prototype, 1994. <http://now.cs.berkeley.edu/Nram/network-ram.html>.
- [19] Evangelos P. Markatos and Manolis G.H. Katevenis. Telegraphos: High-Performance Networking for Parallel Processing on Workstation Clusters. In *Proceedings of the Second International Symposium on High-Performance Computer Architecture, (HPCA)*, San Jose, CA, USA, February 1996.
- [20] Gary Newman. Organizing Arrays for Paged Memory Systems. *Communications of the ACM*, 38(7):93–110, July 1995.
- [21] Mendel Rosenblum and John Ousterhout. The Design and Implementation of a Log-Structured File System. In *Proc. 13-th Symposium on Operating Systems Principles*, pages 1–15, October 1991.
- [22] B.N. Schilit and D. Duchamp. Adaptive Remote Paging for Mobile Computers. Technical Report CU-CS-004-91, University of Columbia, 1991.
- [23] Dolphin Interconnect Solutions. DIS301 SBus-to-SCI Adapter User's Guide.
- [24] A. S. Tanenbaum. *Computer Networks*, chapter 3, page 128. Prentice Hall International, 1989.
- [25] C.A. Thekkath, H.M. Levy, and E.D. Lazowska. Efficient Support for Multicomputing on ATM Networks. Technical Report 93-04-03, Department of Computer Science and Engineering, University of Washington, April 12 1993.
- [26] T. von Eicken, D. E. Culler, S. C. Goldstein, and K. E. Schauser. Active Messages: a Mechanism for Integrated Communication and Computation. In *Proc. 19-th International Symposium on Comp. Arch.*, pages 256–266, Gold Coast, Australia, May 1992.

Biographical information

Evangelos P. Markatos is an Assistant professor at ICS-FORTH and at the University of Crete. He received his diploma in Computer Engineering from the University of Patras in 1988, and the MS and Ph.D. degrees from the University of Rochester in 1990 and 1993 respectively. His interests include parallel and distributed systems, operating systems and computer architecture.

George Dramitinos is a graduate student in Computer Science at the University of Crete, where he received a B.Sc. degree. He has worked at A.C.R.I. in Lyon, France, participating in the design and implementation of an OSF/1 based operating system for the company's supercomputer. He joined ICS-FORTH in 1993. His interests include operating systems, parallel and distributed programming and computer architecture.

The authors can be contacted at {markatos, dramit}@ics.forth.gr. or at their postal address at Institute of Computer Science (ICS), FORTH, Science and Technology Park of Crete, Vassilika Vouton, P.O. Box 1385, GR 711 10 Heraklion, Crete, Greece.

Availability

The most recent version of the pager along with the test programs are freely distributed using ftp from [ftp.ics.forth.gr:pub/pager](ftp://ftp.ics.forth.gr:pub/pager). More information about the project can be found at <http://www.ics.forth.gr/proj/arch-vlsi/os>.

Solaris MC: A Multi Computer OS

Yousef A. Khalidi, Jose M. Bernabeu, Vlada Matena, Ken Shirriff, Moti Thadani

Sun Microsystems Laboratories

2550 Garcia Avenue

Mountain View, CA 94043 USA.

Abstract

Solaris MC is a prototype distributed operating system for multi-computers (i.e. clusters of nodes) that provides a *single-system image*: a cluster appears to the user and applications as a single computer running the Solaris® operating system. Solaris MC is built as a set of extensions to the base Solaris UNIX® system and provides the same ABI/API as Solaris, running unmodified applications. The components of Solaris MC are implemented in C++ through a CORBA-compliant object oriented system with all new services defined by the IDL definition language. Objects communicate through a runtime system that borrows from Solaris doors and Spring subcontracts. Solaris MC is designed for high availability: if a node fails, the remaining nodes remain operational. Solaris MC has a distributed caching file system with Unix consistency semantics, based on the Spring virtual memory and file system architecture. Process operations are extended across the cluster, including remote process execution and a global /proc file system. The external networks is transparently accessible from any node in the cluster. The prototype is fairly complete—we regularly exercise the system by running multiple copies of an off-the-shelf commercial database system.

1. Introduction

Solaris MC¹ is a prototype operating system for a multi-computer, a cluster of computing nodes connected by a high-speed interconnect. The Solaris MC operating system provides a single system image, making the cluster look like a single machine to the user, to applications, and to the network. By extending operating system abstractions across the cluster, Solaris MC preserves the existing Solaris ABI/API and runs existing Solaris 2.x applications and device drivers without modification.

1. Solaris MC is an internal name of a research project at Sun Microsystems Laboratories. More information on the project can be obtained from <http://www.sunlabs.com/research/solaris-mc>.

The decision to design a cluster operating system was motivated by trends in hardware technology. Traditional bus-based symmetric multiprocessors (SMP) are limited in the number of processors, memory, and I/O bandwidth that they can support. As processor speed increases, traditional SMPs will support an even smaller number of CPUs. Powerful, modular, and scalable computing systems can be built using inexpensive computing nodes coupled with high-speed interconnection networks. Such clustered systems can take the form of loosely-coupled systems, built out of workstations [1], massively-parallel systems (e.g. [24]), or perhaps as a collection of small SMPs interconnected through a low-latency high-bandwidth network.

The key to using clustered systems is to provide a single-system image operating system allowing them to be used as general purpose computers. Cluster systems in the past have been mostly used for custom-built parallel and distributed applications, and sometimes as specialized database systems. However, to fully exploit the potential of clustered systems, we believe that they have to be usable as *general purpose computers*, running existing applications without modification. Moreover, clustered systems have to be easy to administer and maintain. The fact that the computer is actually built out of multiple computing nodes should be invisible to the user. Finally, since clustered systems are built out of many components, the clustered system should be *highly-available* and should be able to tolerate the failure of any one component.

Our goals are to make a cluster of nodes that may or may not share memory appear as a single general purpose multiprocessor. It should be seen as a single machine by applications, users and administrators. We want this while preserving object code compatibility (the ABI), minimizing changes to kernel code,

requiring minimal or no change to device drivers, and supporting high availability.

Solaris MC has several interesting features. It:

- Extends existing Solaris operating system

Solaris MC is built on top of the Solaris operating system. Most of Solaris MC consists of loadable modules extending Solaris and minimizes the modifications to the existing Solaris kernel. Thus, Solaris MC shows how an existing, widely used operating system can be extended to support clusters.

- Maintains ABI/API compliance

Existing application and device driver binaries run unmodified on Solaris MC. To provide this feature, Solaris MC has a global file system, extends process operations across all the nodes, allows transparent access to remote devices, and makes the cluster appear as a single machine on the network.

- Supports high availability

The Solaris MC architecture provides fault-containment at the level of an individual node in the multi-computer. Solaris MC runs a separate kernel on each node. A failure of a node does not cause the whole system to fail. A failed node is detected and system services are reconfigured to use the remaining nodes. Only the programs that were using the resources of the failed node are affected by the failure. Solaris MC does not introduce new failure modes into UNIX.

- Uses C++, IDL, and CORBA in the kernel

Solaris MC illustrates how the CORBA object model can be used to extend an existing UNIX to a distributed OS. At the same time it also shows the advantages of implementing strong interfaces for kernel components by using the IDL interface definition language. Finally, Solaris MC illustrates how C++ can be used for kernel development, coexisting with previous code.

- Leverages Spring technology

Solaris MC illustrates how the distributed techniques developed by Spring OS [15] can be migrated into a commercial operating system. Solaris MC imports from Spring the idea of using a CORBA compliant object model [18] as the communication mechanism, the Spring virtual memory and file system architecture [7, 10, 9], and the use of C++ as the implementation language.

One can view Solaris MC as a transition from the centralized Solaris operating system toward a more modular and distributed OS like Spring.

Solaris MC uses ideas from earlier distributed operating systems such as Sprite [19], LOCUS [20], OSF/1 AD TNC [26], MOS [2], and Spring. One key difference from other systems is that Solaris MC shows how a commercial operating system can be extended to a cluster while keeping the existing application base. In addition, Solaris MC uses an object-oriented approach to define new kernel components. Solaris MC also has a stronger emphasis on high availability. Finally, Solaris MC uses new techniques for making the cluster appear as a single machine to the external network.

The remainder of this paper is structured as follows. Section 2 explains the global file system. Section 3 describes how process management is globalized. Section 4 explains how I/O devices are made global and Section 5 explains how network operations are made transparent. Section 6 discusses the object-based communication model of Solaris MC, and explains CORBA and IDL. Section 7 briefly describes how Solaris MC provides high availability. Section 8 provides the current status of Solaris MC, Section 9 compares Solaris MC to other distributed operating systems, and Section 10 concludes the paper.

2. Global File System

Solaris MC uses a global file system to make file accesses location transparent—a process can open a file located anywhere in the system and processes on all nodes can use the same pathname to locate a file. The global file system uses coherency protocols to preserve the UNIX file access semantics even if the file is accessed concurrently from multiple nodes. This file system, called the proxy file system (PXFS), is built on top of the existing Solaris file system at the *vnode* [11] interface. This interface allows PXFS to be implemented without kernel modifications. The PXFS file system provides extensive caching for high performance using the caching approach from Spring [7], and provides zero-copy bulk I/O movement to move large data objects efficiently. This section discusses these features of PXFS in more detail.

PXFS interposes on file operations at the *vnode*/VFS interface and forwards them to the *vnode* layer where the file resides, as shown in Figure 1. Besides files, PXFS also provides access to other types of *vnodes*, such as directories, symbolic links, special devices, streams, swap files, fifos, and Solaris doors². Because PXFS is built on top of the existing file

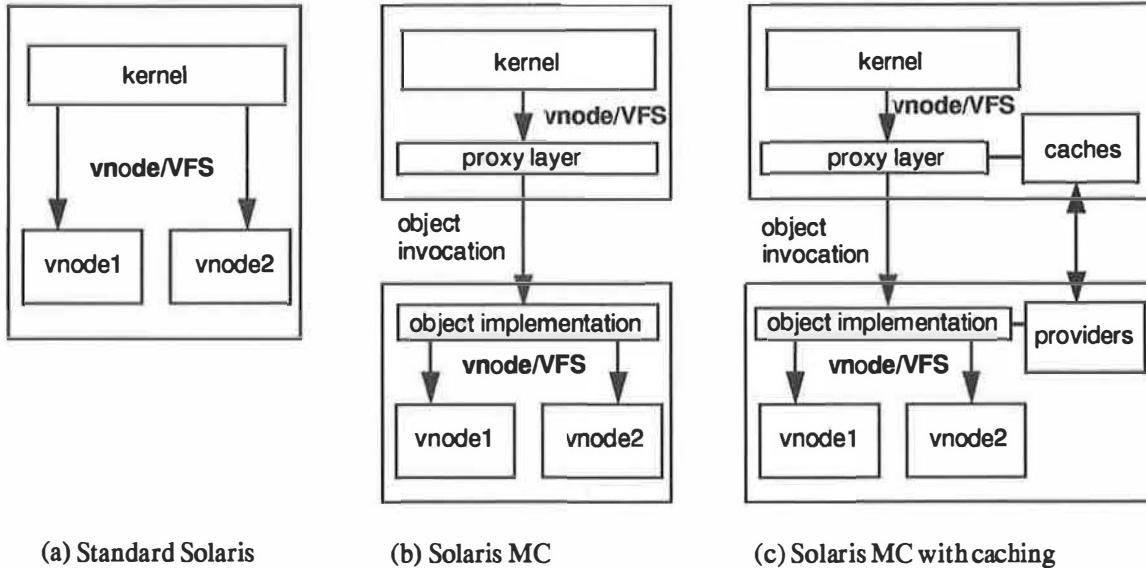


Figure 1. Extending File System Interfaces for Solaris MC. (a) In Solaris, the kernel accesses files through the VFS/vnode operations. (b) In Solaris MC, the VFS/vnode operations are converted by a proxy layer into *object invocations*. The invoked object may reside on any node in the system. The invoked object performs a local VFS/vnode operation on the underlying file system. Neither the kernel, nor the existing file systems have to be modified to run under Solaris MC. (c) Caching is used in Solaris MC to improve performance. Solaris MC supports caching of file pages, directory information, file attributes, and mount points.

system, it can leverage off the existing file system code. This is an important difference from distributed file systems such as Sprite or Spring that rewrite the entire file system.

PXFS uses extensive caching on the clients to reduce the number of remote object invocations. Figure 2 shows the objects used in the file paging and attribute caching protocols. The design of PXFS was influenced by the Spring file system and its caching architecture [7, 17, 16]. A client cache is implemented through a *cached* object on the client to manage the cached data and a *cacher* object on the server to maintain consistency. For data, the client has a *memcache* object and the server has a *mempager* object. For attributes, the client has a *attrcache* object and the server has a *attrprov* object.

As an example, suppose a process on Client 1 wishes to page in a page from a file. A *memcache* is a vnode in addition to being an IDL object, so it can accept GETPAGE and PUTPAGE operations from the Solaris virtual memory system. The memcache vnode is used as the paged vnode for the VOP_MAP operations on the proxy vnode. Memcache searches the local cache for the page. If it is not available,

2. Solaris doors is a new IPC mechanism in Solaris 2.5 that is based on the Spring IPC mechanism [15].

memcache requests the page from the associated mempager. The mempager checks the other mempages to see if another client has the page, to maintain consistency. Finally, the page is obtained from the backing server vnode. Thus, PXFS has control over global page coherence.

The PXFS coherency protocol is token-based and allows a page to be cached read-only by multiple caches or read-write by a single cache. If a dirty page is transferred from one node to another, it is first written to the stable storage on the server to avoid losing updates due to crashes of unrelated nodes. Similarly, an attribute cache is also protected by a reader-writer token. The token is also used to enforce atomicity of read/write system calls on regular files. Token management is integrated with data transfer for better performance.

Directory caching and caching of mount points is done in a fashion similar to attribute caching. Directory operations that create or remove objects are implemented as write-through to be reflected synchronously in stable storage on the server.

PXFS has a “bulkio” object handler to perform zero-copy transfers between nodes of large data (file pages, uioread/uiowrite data) if the hardware interconnect has sufficient support. For example, if a

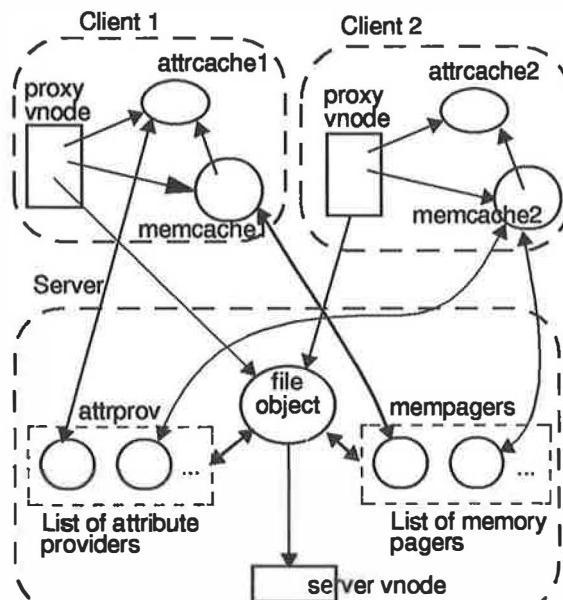


Figure 2. File Paging and Attribute Caching. Each client has a memcache object to cache data and an attrcache object to cache attributes. The server has corresponding mempager and attrprov objects to provide the data and attributes. The file object is an IDL object implementing the file protocol. The server vnode provides the underlying file storage.

process takes a page fault, it allocates a page in the local cache and invokes the *page_in* method on the mempager. The server then allocates a kernel buffer and reads the data from the disk into the buffer. The data is then transferred using the bulkio handler directly into the page on the client. If the underlying hardware supports shared memory, the server can map the client page and read data from the disk directly into the page without the need for an intermediate buffer on the server. By using a separate handler for bulk I/O, no changes to the PXFS client or server code are necessary to port PXFS to a different interconnect; only the bulkio handler has to be ported to take full advantage of the hardware.

3. Global Process Management

Global process management in Solaris MC extends OS process operations so that the location of a process is transparent to the user. While the threads of a single process must be on the same (possibly multiprocessor) node, a process can reside on any node. The design goals of process management are to support POSIX semantics for process operations while providing good performance, supplying high availability, and minimizing changes to the existing Solaris kernel. This section discusses the implementation of process management and how it

transparently provide signals on global process ids, distributed waits, the /proc file system, and process migration.

Process management is implemented in a kernel module above the existing Solaris kernel code that manages the global view of processes. As illustrated in Figure 3, this layer consists of a virtual process (vproc) object for each local process and a node manager for the node. The vproc maintains state such as the parent and children of the process. The node manager keeps track of the local processes and the other nodes. Additional objects manage process groups and sessions.

The global process layer interacts with the rest of the system in several ways. First, process-related system calls are redirected to this layer. Second, a small number of hooks were added to the kernel to call this layer when appropriate. Finally, the vproc layers on different nodes communicate through IDL interfaces. Process management was made more difficult by the lack of an existing kernel interface (analogous to vnodes for the file system). We are exploring if the vproc interface can be extended to a flexible kernel interface useful for other system extensions.

Process identifiers (pids) in Solaris MC use a single global pid space and encode the home node of the process in the top bits. Thus, an arbitrary process can be located from its pid by contacting the home node, which knows the current location; this location can then be cached. The signal delivery code, for instance, uses the pid to deliver signals to a process no

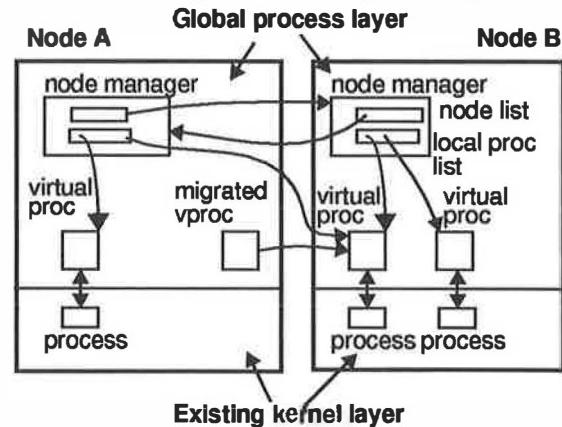


Figure 3. The data structures of the global process layer. Each node has a node manager object that has a list of all processes created or residing on the node and a list of the other nodes. Each process has a virtual process (vproc) object associated with it. When a process migrates, the old vproc is left behind to forward any operations. The vprocs keep track of the parent/child relationships of the processes.

matter where it resides. The pid encoding also ensures that processes on different nodes will not be created with the same pid. The same pid is used inside and outside the kernel; Solaris MC does not use distinct local (internal) pids and global (external) pids.

Waits pose problems for a cluster because a parent and child may be on separate nodes. In Solaris MC, distributed waits are implemented by having the child inform the parent of each state change (exit, stopped, continued, or debugged). The parent keeps track of the state of each child and wait operations use this local copy.

In Solaris, the /proc pseudo file system provides access to each process in the system; this is used by ps and the debugger, for instance. In Solaris MC, the /proc file system is extended to cover all processes in the cluster. Code for /proc in the pxfs file system merges together local /proc_local file systems into a global /proc. Thus, directory operations on /proc show the process entries in all the local /procs, and lookup operations are redirected to the appropriate node.

Solaris MC currently supports remote execution of processes and will soon support remote forks and migration of existing processes. For a remote fork or migration, most of the process's state will be moved automatically through the consistency mechanism of pxfs. A "shadow vproc" is left behind when a process migrates; any operations received by the shadow vproc are forwarded to the vproc on the node where the process resides. The policy decisions on load balancing will be built on top of the migration mechanisms; one possibility is to use a migration daemon, as in Sprite [5], that will decide which nodes should receive processes. However, we believe that the main use of process migration will be for planned shutdown of cluster nodes, rather than fine load balancing across the cluster; load balancing will largely be managed by placement of processes at exec time.

Global process management in Solaris MC will support high availability. That is, the failure of a node will not interfere with processes on another node. While the processes on a failed node will die, the rest of the system will continue, after a recovery phase. Parents and children will be notified appropriately of process failures. A new node will take over as home node for the failed node and migrated processes that originated on the failed node will now use the new node as home.

4. I/O Subsystem

The I/O subsystem makes it possible to access any I/O device from any node in the multi-computer without

regard to the physical attachment of devices to nodes. Applications are able to access I/O devices as local devices even when the devices are physically attached to a node different from the one on which the application is running. Several areas require attention to ensure this access:

- **Device configuration:** Solaris provides dynamically loadable and configurable device drivers. Solaris MC transparently provides a consistent view of device configurations through a distributed device server that is notified when a new device is configured into the system on a particular node. When the device driver corresponding to the newly configured device is invoked on a different node, it is loaded on that node using the DDI/DKI device interfaces defined for Solaris. Different nodes in the system may have different devices attached and different sets of drivers/modules loaded in kernel memory at any point in time.

The device server distributes the functionality of the Solaris *modctl()* interface, which handles the loading and unloading of dynamically loadable modules. Module configuration routines such as *make_devname()* add the new device names to the device server. Module control interfaces such as *mod_hold_dev_by_major()*, *ddi_name_to_major()*, and *ddi_major_to_name()* look-up the distributed device database rather than local data structures.

- **Uniform device naming:** Device numbers provide information about the location (i.e. node number) of the device in the system in addition to the type of device and the instance or unit number of the device. The operating system associates a location with every device special file. When a device is opened, the *open()* is directed to the node to which the physical device is attached.
- **Providing process context to device drivers:** Device drivers require access to process context for data transfer and credentials checking. In Solaris MC, the calling process may be on a different node than the node on which the driver executes. Consequently, the process context in which the driver runs is different from the process context of the calling process. The operating system provides a logical equivalence between the two processes in order for device drivers to be able to function without modification.

The Streams framework poses additional problems (which are not discussed in detail here due to space limitations). Solaris MC allows Streams device

drivers and modules that use procedural interfaces to work unchanged in the new environment. Some modules, however, do not strictly obey the Streams interface; they may either be modified to run on Solaris MC, or they may be confined to one node in the cluster.

5. Networking

The networking subsystem in Solaris MC creates a single system image environment for networking applications. The operating system ensures that network connectivity is the same for every application regardless of which node the application runs on. This goal is achieved with minimal impact on the existing network subsystem implementation and without any changes to applications.

We considered three approaches for handling network traffic. The first approach was to perform all network protocol processing on a single node. This approach, however, is not scalable to large numbers of nodes. The second approach was to run network protocols over the interconnection backplane. This approach requires each node to have a separate network address, which prevents transparency. The third approach, which we took, was to use a packet filter to route packets to the proper node and perform protocol processing on that node.

Our approach creates the illusion that the set of real network interfaces available in the system is local to each node in the system. Applications are unaware of the real location of each network device and their view of the network is the same from every node in the system. When an application transmits data over an illusory network device on a node, the framework forwards the outgoing network packet to the real device. Similarly, on the input side, the framework forwards packets from the node on which the real network device is attached to the node where the appropriate application is running.

The advantages of our design are (a) protocol processing is not limited to those nodes that have network devices, (b) only one new module is written to handle networking for most protocol stacks, and (c) changes to the protocol stacks are minimized.

There are three key components of the Solaris MC networking subsystem:

- Demultiplexing of incoming packets to the “correct” node: Incoming packets are first received on the node that has the network adapter physically attached to it. The data may, however, be addressed to an application running on a

different node. Solaris MC includes an enhanced implementation of the programmable Mach packet filter [14, 25], which extracts relevant information from each packet and matches it against state information maintained by the host system. Once the destination node within the multi-computer system is discovered, the packet is delivered to that node over the system interconnect.

- Multiplexing of outgoing packets from various nodes onto a network device: All protocol processing for outgoing packets is performed on the node on which the endpoint for the network connection exists. The layer that passes data to the device driver makes use of remote device access (transparently) to send data over the physical medium.
- Global management of the network name space: Network services are accessed through a service access point or sap. (For TCP/IP, the saps are simply ports.) Providing a single system image of the sap name space requires coordination between the various nodes. In Solaris MC, a database that maps service access points to nodes within the multi-computer is maintained by the SAPServer, which ensures that the same sap is not simultaneously allocated by different nodes in the system.

The structure of the networking system is shown in Figure 4. The *mc_net* module is the packet filter that creates the illusion of a local lower stream corresponding to a remote physical network device in the system. The *mc_net* module is pushed above the cloneable network device driver by the Solaris MC network configuration utilities. The network stack, with the exception of the *mc_net* module is oblivious of the location of the network device within the multi-computer system. In the figure, the SAPServer is shown independent of a node for clarity; in reality it is provided on one or more of the nodes of the system.

Solaris MC networking also provides the ability to replicate network services to provide higher throughput and lower response times. This is achieved by extending the API to allow multiple processes to register themselves as servers for a particular service. The network subsystem then chooses a particular process when a service request is received. For example, *rlogin*, *telnet*, and *http* servers are by default replicated on each node. Each new connection to these services is sent to a different node in the cluster based on a load balancing policy (currently, a simple round-robin load distribution policy). This allows the

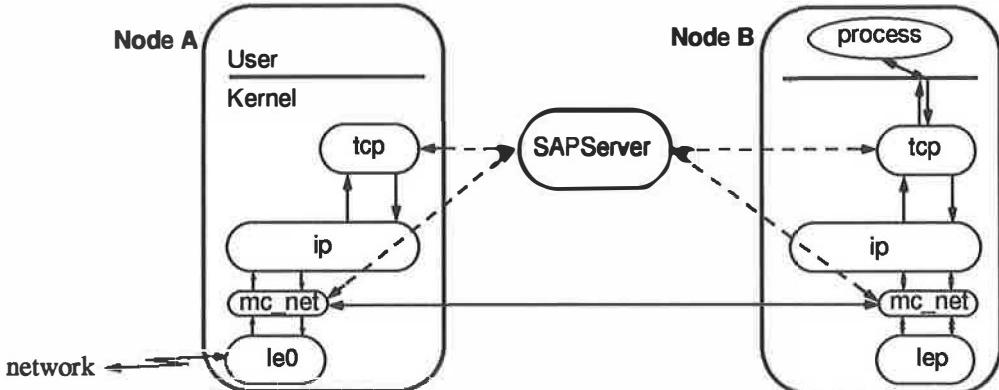


Figure 4. Multi-computer Networking Set-up. The mc_net packet filter makes the le0 network device appear local to the application process. TCP/IP protocol processing occurs on node B, preventing node A from becoming a bottleneck. Solid lines show data traffic and dotted lines show service access port control communication.

cluster to be used as a HTTP server, for example, with all nodes handling requests in parallel.

Other features of the Solaris MC networking subsystem are management of global state in the network protocols, such as network statistics maintained for network management agents and network state information acquired from routers or peers on the network. In the former case, the network management agents are modified to collect information from all the component nodes of the multi-computer, while in the latter case information collected on any node is broadcast to the other nodes.

6. Communication & Programming Infrastructure

Solaris MC is built from a set of components on top of the Solaris basic kernel. Those components include most OS services, from file system support to global process management and networking management. The programming and communications framework provides support for implementing the components and the communication between components. The framework includes a programming model, a compiler, and run time support for component implementation.

6.1 Programming model

Solaris MC components require a mechanism for accessing them both locally and remotely, and to determine when a component is no longer used by the rest of the system. At the same time, it is essential that each new component have a clearly specified interface, permitting its maintenance and evolution. These two requirements led us to decide on the adoption of an object oriented approach to the design of Solaris MC.

From the available possibilities we decided to adopt the CORBA (*Common Object Request Broker Architecture*) [18] object model, as the best suited for our purposes. CORBA is an architecture with mechanisms for objects to make requests and receive responses in a heterogeneous distributed environment, somewhat similar to RPCs. CORBA provides a strong separation between interfaces and implementations. In CORBA, an interface is basically a set of operations and each object accepts requests for the operations defined by the associated interface. How a given object implements an interface is up to the implementor of the particular object. CORBA also includes reference counting. In order to perform a request on an object, the client code must obtain a reference to that object, allowing the system to keep track of the number of references.

Interfaces are defined by using CORBA's *Interface Definition Language* (IDL) [23]. IDL allows the definition of interfaces by specifying the set of operations the interface accepts (similar to C function declarations), as well as the set of exceptions any given operation may raise. Interfaces can be composed using *interface inheritance* mechanisms, including multiple inheritance. Client and server object implementation code can be written using any programming language for which a mapping from IDL has been established. Currently there are a few such programming languages, including C and C++. We decided to use C++ as it provided the best match for the CORBA object model.

Every major component of Solaris MC is defined by one or more IDL specified object type. All interactions among the components are carried out by issuing requests for the operations defined in each component's interface. Such requests are carried out

independently of the location of the object instance by using our own ORB (*Object Request Broker*), or runtime. When the invocation is local (within the same address space), it proceeds like a procedure call. When the invocation crosses domains (across address spaces or nodes), the invocation proceeds essentially as an RPC, where the client code uses stubs, and the server (implementation) code uses skeleton code to handle the call.

The stubs used by the client code, as well as the skeletons used by the server code are generated automatically from the IDL interface definition by a CORBA IDL to C++ compiler. We currently use a modified version of the Fresco IDL to C++ compiler [13].

6.2 The Run time System

The different components of the system communicate using the services of the ORB. The main functions of the Solaris MC ORB are reference counting, marshaling/unmarshaling support, remote request support (RPC), and communication fault recovery. The three main goals of our ORB architecture are to provide an efficient object invocation mechanism, easy configuration of clusters, and support for high availability.

Solaris MC's ORB is composed of three layers: the handler, the xdoor, and the transport layer (Figure 5). Each object reference is associated with a handler. The handler is responsible for preparing inter-domain requests to the object whose reference it handles. A handler is also in charge of performing marshaling of its associated references, as well as of local (to an address space) reference counting. The handler layer implements the subcontract paradigm [21], providing flexible means of introducing multiple object manipulation mechanisms, such as zero-copy.

In order to perform an invocation on its object, a handler is associated with one or more *xdoors*. The xdoor layer implements an RPC-like inter process communication mechanism. This layer extends and builds on the functionality of the Solaris doors mechanism. With xdoors, it is possible to perform arbitrary inter-domain (intra- or inter-node) object invocations following an RPC scheme. References to xdoors are carried out with invocations and replies, permitting the ORB to locate particular instances of implementation objects. The xdoor layer implements a reference counting mechanism for ORB and application structures. Together, the handler and xdoor layers support

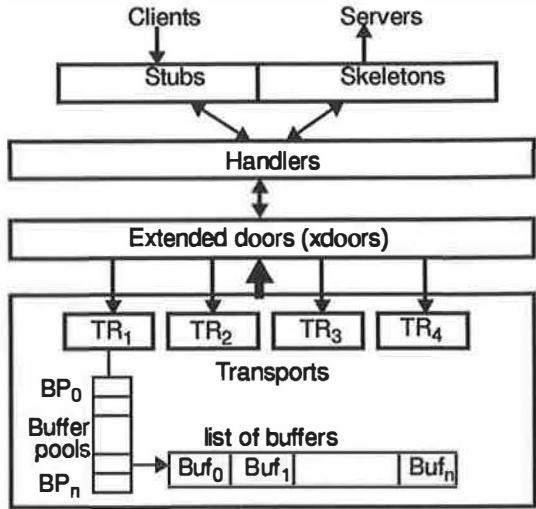


Figure 5. The different layers of the ORB.

efficient parameter passing for inter-address-space, inter- and intra-node invocations.

The transport layer defines the interface that a transport (such as Ethernet or Myrinet [4]) has to satisfy to be used by the xdoor layer. Transports implement reliable *sends* of arbitrary length messages. It is also possible to register receive functions with a transport. A transport has a set of buffer pools with lists of buffers, waiting to be filled by incoming messages. Arriving messages are stored in buffers from the buffer pools specified in the message's headers. Thus, the transport's interface is both simple and sufficiently powerful to support highly efficient object invocation, providing message delivery with scatter and gather capabilities through the buffer pools. A Solaris MC system can support several ORB transports at the same time. While our main goal is to support closely connected clusters, this capability makes it easy to configure the system to have some long distance links within the cluster.

Together, the three layers provide support for efficient parameter marshaling and passing, making it possible to implement zero-copy schemes for inter-node communications when the communications hardware supports it.

To support high availability, the xdoor layer never aborts an outstanding invocation unless a failure in the cluster is detected. In that case, outstanding invocations to failed nodes are aborted, and an exception is raised which can be caught by the handler layer, or by the component code itself. Services needing high availability make use of the information in the exception either directly or by means of special handlers. In addition to this failure reporting, the xdoor layer

implements a reference counting algorithm that can recover after node failures. For efficiency, the algorithm is optimistic in nature, performing most of its work when an actual node failure is detected.

The ORB permits both kernel- and user-level communication. The ORB is implemented as a loadable kernel module to be used by the code residing in the kernel, and as a library, to be used by code executing in a user-level process. Most of the code used in both cases is identical, differing mostly in the xdoor and transport layers. Calls which target objects served by a user-level process are routed through the kernel xdoors.

6.3 C++ in the kernel

All Solaris MC extensions are implemented in C++, and are incorporated into a Solaris kernel as loadable modules. In order to mix the C++ code with the existing kernel, it was necessary to create a special loadable module containing the C++ runtime support. The basic task of the loadable module is to provide some new relocation types to the kernel dynamic linker. It also supports the "new" and "delete" operators, as well as the C++ exception handling mechanism, which in turn is used to implement our ORB's exceptions.

So far we have had no problems with code size or speed, finding no major difference with C compiled code. We use C++ not only for writing the components defined with the IDL interfaces, but also any other code performing auxiliary or complementary functions. We make conservative use of C++ features, only using those with sufficient advantages for their cost. Thus, we do not use virtual base classes (the current compiler implementation makes them very space-inefficient), or return objects by value. On the other hand, we find C++ exceptions extremely useful. Exceptions are extensively used throughout our code to signal abnormal conditions and errors.

7. Support for High Availability

Solaris MC integrates the support for high availability into the operating system. Solaris MC divides the responsibility for high availability into several layers: failure detection and membership service, object and communication framework, reconfiguration of system services, and reconfiguration of user level programs. A more complete description of the high availability support in Solaris MC will be described in a forthcoming technical report. Here we provide a brief description of the architecture and some examples.

At the lowest level, a cluster membership monitor (CMM) detects a communication or node failure. The

CMM informs the ORB that a cluster reconfiguration is in progress. The CMM then uses a distributed membership protocol to reach a global agreement on the current cluster configuration. Once an agreement is reached, the ORB is contacted. In turn, the ORB invalidates all client xdoors for objects residing in the failed node. Furthermore, the ORB runs a reference recovery protocol which removes all object references held by processes on the failed node.

Distributed system services can learn about a failure in two ways. First, they may get an exception indicating the failure of the node servicing a currently invoked object reference. Second, they can use special handlers which are notified when the xdoor they selected to perform an invocation has been invalidated as a result of a failure. For example, when a node caching file pages crashes, the pager object receives an *unreference* upcall. The pager then cleans up any locks held by the cache, allowing other nodes to access the file.

Figure 6 illustrates one system configuration to make Solaris MC system services highly available. The figure shows a four node system configured as an NFS server connected to an Ethernet. A file system is stored in a disk volume mirrored on two disks. The disks are dual ported and connected to a pair of nodes. One node is designed as the primary server for the file system (Node 3); the other is a backup (Node 4). A journaling file system is used to minimize file system recovery time. The file system is exported via NFS to remote clients (C1). The file is also accessed by processes P1 and P2 running locally on the cluster. The network interface *if1* is the primary interface for the cluster IP address; *if2* is a backup.

If Node1 crashes, Solaris MC relocates the IP address to *if2* on Node 2. The crash is transparent to P2 and the remote NFS client. If Node 3 crashes, the backup volume manager on Node 4 takes over and recovers the disk volume. The backup IDL objects on Node 4 continue to serve the file system. The crash and takeover are transparent to P1, P2, and any remote client.

A special object handler is used to implement the file objects on Node 3 to facilitate transparent failover to the backup object. The special handler stores enough information in the object reference to perform a switchover from a primary object on Node 3 to a backup object on Node 4. The switchover is transparent to the holder of the object reference (e.g. a pefs proxy vnode). In the example we assume that a journaling file system makes changes to the file non-vola-

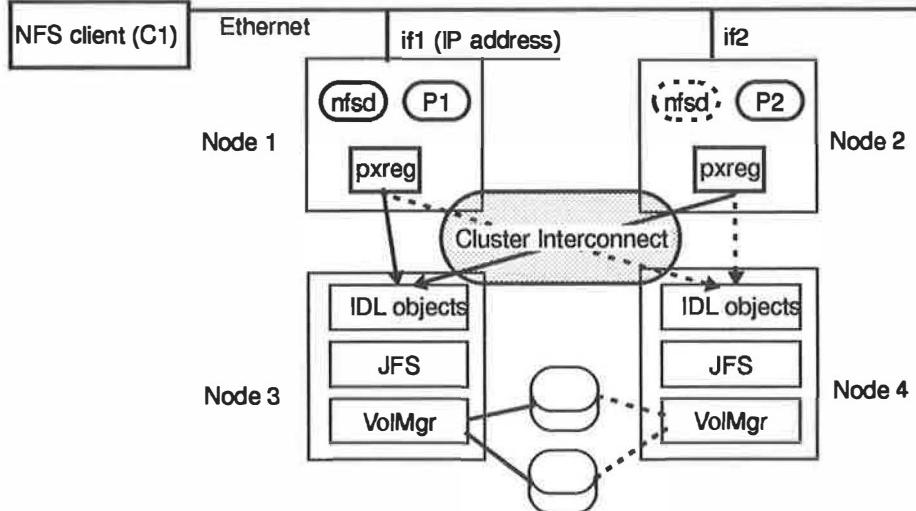


Figure 6. An example of high availability in Solaris MC. The system is configured as a NFS server with redundancy.

tile state recoverable. As for the volatile state (i.e., locks and tokens), there are basically three strategies to preserve it across a takeover by the backup node: backup solicits the volatile state from clients during takeover (long takeover), backup has an up-to-date volatile state (high runtime overhead), or the volatile state is in stable storage (high runtime overhead if no hardware support).

The focus on high availability differentiates Solaris MC from layered cluster software products, such as AT&T LifeKeeper, IBM HACMP 6000, HP MC/Service Guard, and Sun SPARCcluster PDB. Solaris MC is much easier to configure for high availability than the layered products, making the administration model scalable to a high number of nodes. Tight integration of key high availability protocols (such as the node membership protocol) with the low level communication framework results in faster failure detection and recovery than is achievable in layered systems.

8. Status

Most of the architecture described in this paper has been implemented: communication and object support, including the ORB and object reference counting; C++ support in the kernel; the global file system; most of process management, including remote exec, wait, signals, and the /proc file system; global networking support for TCP/IP, including extensions to the API to allow more than one server to service incoming connections on the same port; access to remote I/O devices, with no modifications to device drivers; a group membership and status

monitor; and a set of extensible performance evaluation tools.

The system was initially developed on Solaris 2.4 and has since been moved to Solaris 2.5 with little effort. The prototype is fairly complete—we regularly exercise the system by running parallel makes and multiple copies of a commercial database server.

All implementation work is done in C++, and all new interfaces are defined using IDL. With the exception of a few minor changes to the kernel proper, the bulk of Solaris MC extensions consist of a set of loadable modules or user-land servers.

The hardware prototype currently consists of sixteen dual-processor SparcStation 10 and 20 machines, partitioned into two or more clusters. The developers' workstations act as front-end machines to the Solaris MC cluster. We use a variety of networks as the system interconnect, including 100baseT ethernet and Myrinet [4].

9. Related Work

Solaris MC uses ideas from earlier distributed operating systems such as Chorus Mix, LOCUS, MOS, OSF/1 AD TNC, Spring, Sprite, and VAXclusters. There are, however, significant differences in our approach, compared to previous systems:

- Solaris MC shows how a commercial operating system can be extended to a cluster while keeping the existing application base.
- Solaris MC emphasizes high availability.

- Solaris MC uses an object-oriented design. The system was built with the Corba object model.
- Solaris MC uses new ideas from Spring, especially filesystem and virtual memory ideas.

For example, unlike systems such as Spring, and Sprite, Solaris MC builds on a commercial operating system while maintaining binary compatibility with a large existing application base. Also, most of the other systems, with the notable exception of VAXclusters, do not emphasize high availability. Finally, Solaris MC introduces object-oriented techniques based on the Corba object model, and builds on the experience of the Spring system.

10. Conclusions & Future Work

We have built a prototype operating system that provides a single-system image for distributed tightly-coupled hardware systems. The prototype consists of a set of extensions to a commercial operating system. The resultant system thus leverages all existing applications of the OS and enables the OS to extend to a new class of computers. By extending an existing operating system, rather than writing an entirely new one, we were able to leverage off the existing OS code base and device drivers, dramatically reducing the development effort.

We made the early decision to build our system using the Solaris system in order to leverage the large investment in application and system software. Several Solaris features made our job easier, including loadable modules and dynamic linking, both for the kernel and user processes; the basic system VFS and DDI interfaces; the multi-threaded architecture of the system; and the new door IPC mechanism. On the other hand, there are portions of the system that are difficult to extend to a clustered system, including the STREAMS framework because many existing modules do not adhere to the framework and assume that they are running on a shared memory system. Some UNIX semantics are also difficult (but not impossible) or are inefficient to extend to a clustered system, including POSIX controlling terminal and sessions semantics, file descriptor sharing, and fork semantics.

Our experience so far with the use of IDL and CORBA to design and implement Solaris MC is very positive. The use of IDL gives us a collection of clearly defined interfaces, and the IDL to C++ translator conveniently creates the glue we need to perform arbitrary service requests from our components. An

additional advantage of using IDL is the little effort it took us to adapt Spring technology to Solaris MC.

The structure of Solaris MC ORB allows us to create special handlers to efficiently transmit bulk data and other user-defined structures between nodes. These handlers make use of the transport layer to avoid extra copying of large amounts of data.

On the other hand, we find the CORBA model lacking in two areas. First, there is no straightforward way to perform asynchronous object invocations. Secondly, there is no support for performing group invocations to a set of objects supporting a common interface, a feature that would be useful for the cluster membership monitor. We had to go outside the CORBA model for group multicast.

More work remains for the future. High availability support needs more work. We plan to port a volume manager and provide more support for system administration. We also plan to move to faster interconnect hardware, and to measure and analyze the system performance. Finally, we plan to experiment with heterogenous clusters.

Acknowledgments

We would like to acknowledge Madhu Talluri, Remzi Arpacı, Francesc Munoz Escoi, and Aman Singla for their contributions to the project.

References

- [1] T.E. Anderson, D.E. Culler, D.A. Patterson, "A Case for NOW (Networks of Workstations)," *IEEE Micro*, February, 1995.
- [2] A. Barak and A. Litman, "MOS: A Multicomputer Distributed Operating System," *Software—Practice & Experience*, vol. 15(8), August 1985.
- [3] N. Batlivala, *et al.*, "Experience with SVR4 over CHORUS," *Proceedings of USENIX Workshop on Microkernels & Other Kernel Architectures*, April 1992.
- [4] N. J. Boden, D. Cohen, R.E. Felderman, A.E. Kulawik, C.L. Seitz, J.N. Seizovic, W. Su, "Myrinet: A Gigabit-per-Second Local Area Network," *IEEE Micro*, February 1995.
- [5] F. Douglis and J. Ousterhout, "Transparent Process Migration: Design Alternatives and the Sprite Implementation," *Software—Practice & Experience*, vol. 21(8), August 1991.
- [6] Intel Corporation, *Intel Paragon XP/S Supercomputer Spec Sheet*, 1992.

- [7] Yousef A. Khalidi and Michael N. Nelson, "Extensible File Systems in Spring," *Proceedings of the 14th Symposium on Operating Systems Principles (SOSP)*, December 1993.
- [8] Yousef A. Khalidi and Michael N. Nelson, "An Implementation of UNIX on an Object-oriented Operating System," *Proceedings of Winter '93 USENIX Conference*, January 1993.
- [9] Yousef A. Khalidi and Michael N. Nelson, "The Spring Virtual Memory System," Sun Microsystems Laboratories Technical Report SMLI-TR-93-09, February 1993.
- [10] Yousef A. Khalidi and Michael N. Nelson, "A Flexible External Paging Interface," *Proceedings of the Usenix conference on Microkernels and Other Architectures*, September 1993.
- [11] Steven R. Kleiman, "Vnodes: An Architecture for Multiple File System Types in Sun UNIX", *Proceedings of '86 Summer Usenix Conference*, pp. 238-247, June 1986.
- [12] N. Kronenberg, H. Levy, and W. Strecker, "VAX-clusters: A Closely-Coupled Distributed Systems," *ACM Transactions on Computer Systems*, vol. 4(2), May 1986.
- [13] Mark Linton and Douglas Pan, "Interface Translation and Implementation Filtering," *Proceedings of the USENIX C++ Conference*, 1994.
- [14] C. Maeda, B.N. Bershad, "Protocol Service Decomposition for High-performance Networking," *Proceedings of the 14th Symposium on Operating Systems Principles (SOSP)*, December 1993.
- [15] James G. Mitchell, et al., "An Overview of the Spring System," *Proceedings of Compcon Spring 1994*, February 1994.
- [16] Michael N. Nelson, Graham Hamilton, and Yousef A. Khalidi, "A Framework for Caching in an Object-Oriented System," *Proceedings of the 3rd International Workshop on Object Orientation in Operating Systems*, December 1993.
- [17] Michael N. Nelson, Yousef A. Khalidi, and Peter W. Madany, "Experience Building a File System on a Highly Modular Operating System," *Proceedings of the 4th Symposium on Experiences with Distributed and Multiprocessor Systems (SEDMIS IV)*, September 1993.
- [18] Object Management Group, *The Common Object Request Broker: Architecture and Specification*, Revision 1.2, December 1993.
- [19] J. Ousterhout, A. Cherenson, F. Douglis, M. Nelson, and B. Welch, "The Sprite Network Operating System," *IEEE Computer*, February 1988.
- [20] G. Popek and B. Walker, *The LOCUS Distributed System Architecture*, MIT Press, 1985.
- [21] G. Hamilton, M. L. Powell, and J. G. Mitchell, "Subcontract: A flexible Base for Distributed Programming," *Proceedings of the 14th Symposium on Operating Systems Principles (SOSP)*, December 1993.
- [22] Glenn C. Skinner and Thomas K. Wong, "Stacking Vnodes: A Progress Report." *Proceedings of the Summer 1993 Usenix Conference*, 1993.
- [23] Sun Microsystems, Inc. *IDL Programmer's Guide*, 1992.
- [24] Thinking Machines Corp., *The Connection Machine System: CM-5*, 1993.
- [25] M. Yuhara, C. Maeda, B.N. Bershad, J.E.B. Moss, "Efficient Packet Demultiplexing for Multiple Endpoints and Large Messages," *Proceedings of Winter '94 USENIX Conference*, January 1994.
- [26] Roman Zajcew, et al., "An OSF/1 UNIX for Massively Parallel Multicomputers," *Proceedings of Winter '93 USENIX Conference*, January 1993.

Yousef A. Khalidi (yak@eng.sun.com) is currently a Senior Staff Engineer and Principal Investigator at Sun Microsystems Laboratories. His interests include operating systems, distributed object-oriented software, computer architecture, and high speed networking. He is one of the principal designers of the Spring operating system, and a co-winner of Sun's President Award in 1993. He has M.S. and Ph.D. degrees in Information and Computer Science from Georgia Institute of Technology, where he was one of the principal designers of the Ra and Clouds operating systems.

José M. Bernabéu (josep@iti.upv.es) is currently a Senior Staff Engineer at Sun and a Visiting Professor from the Universidad Politécnica de Valencia, Spain, where he heads the Distributed Systems Group, and directs the "Instituto Tecnológico de Informática." His interests include operating systems, distributed algorithms, distributed object-oriented software, shared memory models, and reliability. He received an M.S. in Physics from the University of Valencia in 1982, and M.S. and Ph.D. degrees in Computer Science from the Georgia Insti-

tute of Technology, where he was one of the principal designers of the Ra and Clouds Operating Systems.

Vlada Matena (vlada@eng.sun.com) is a Senior Staff Engineer at Sun Microsystems Laboratories and PI for High Availability. His interests include OS, reliable distributed systems, and database technology. He has been a key contributor to Sun's database server technology as the principal designer of the SPARCcluster Parallel Database, Sun Database Excellerator, SunDBM, and NetISAM products. He received an SMCC Engineering Excellence Award in 1994. He received an M.S. in Computer Science from the Prague Institute of Technology in 1984.

Ken Shirriff (shirriff@eng.sun.com) is a Staff Engineer at Sun Microsystems Laboratories. His interests include operating systems, cryptography, and fractals. He obtained a B.Math degree from the University of Waterloo in 1987 and M.S. and Ph.D. degrees in computer science from U. C. Berkeley where he worked on the Sprite operating system.

Moti N. Thadani (thadani@eng.sun.com) is a Staff Engineer at Sun Microsystems Laboratories. His interests include computer networking and distributed and object systems. Before joining Sun, he worked at Unisys, IBM, and Data General. He received an M.S. in Computer Science from Tulane University and a B.Tech. in Electrical Engineering from the Indian Institute of Technology, New Delhi.

A New Approach to Distributed Memory Management in the Mach Microkernel

Stephan Zeisset, Stefan Tritscher, Martin Mairandres

Intel Corporation

ABSTRACT

In this paper we describe a new approach towards extending Mach virtual memory semantics across the node boundaries of a multicomputer system. At its core, the Advanced Shared Virtual Memory (ASVM) system employs algorithms from the realm of shared virtual memory that were adapted and extended to create a system that is efficient and scalable and supports full VM semantics, paging between node memories and efficient execution of SVM applications. Our performance measurements demonstrate ASVM's superior efficiency and scalability compared to its predecessor, the Extended Memory Manager (XMM) that is part of the Mach kernel's NORMA distribution.

1. Introduction

Multicomputer systems generally run different copies of their operating system on each of their processing nodes. To ease both programming and system administration, the more advanced multicomputer operating systems, among them OSF1/AD TNC [2], support the concept of a single system image. This provides the user with the illusion of a single workstation environment, including single IP, process and file name spaces.

To accomplish this, OSF1/AD TNC relies heavily on two additions to its Mach microkernel that are also known as NORMA (No Remote Memory Access) extensions: NORMA-IPC for extending interprocess communication and XMM for extending virtual memory semantics across the multicomputer's node boundaries. NORMA-IPC is used by the operating system for all kinds of internode communication and synchronization, while XMM provides copy-on-access semantics for remote task creation and shared memory semantics for the memory mapped file system and user created shared memory segments.

Unfortunately, the original design of NORMA-IPC and XMM did not perform well on large scale parallel machines. The main deficiencies of NORMA-IPC

were its broken flow control in many-to-one communication scenarios and its poor utilization of the available communication bandwidth. The main deficiencies of XMM were its non-scalable approach to handling shared memory and an inefficient communication protocol.

These problems prompted Intel's Scalable Systems Division, which based the operating system of its Paragon multicomputer system on OSF1/AD TNC, to redesign the NORMA-IPC and XMM subsystems. In this paper we will concentrate on the XMM rewrite effort and its outcome, the Advanced Shared Virtual Memory (ASVM) system.

2. Background

2.1 The Paragon multicomputer system

The Paragon is a multiprocessor system with distributed memory. Each of its nodes contains two or three i860XP processors that share a local memory with a size of 16 to 128 MBytes. One of the processors usually acts as a dedicated message passing processor while the others execute user tasks. The nodes are interconnected by a two dimensional mesh of worm-hole routed communication channels with a raw bandwidth of 200 MByte/s in each direction. Existing installations reach up to 1792 nodes.

Three properties of the Paragon multicomputer system had a major influence on the design of ASVM: The high bandwidth and low latency interconnect, the great number of nodes a system can contain and the fact that there is typically one node with an attached disk drive for 32 nodes that are used for computation.

2.2 The Mach Kernel's VM system

The Mach kernel's virtual memory (VM) system provides the abstraction of memory objects. These are user managed entities which are represented by a VM object on the kernel side and a memory object port on

the user side. The available physical memory is used as a cache for memory object contents. User level pager tasks are responsible for providing the initial contents of a memory object and for preserving the object's data when it is evicted from the cache. The External Memory Management Interface (EMMI) protocol is used between the kernel and the pager tasks to exchange page contents and access rights (see FIGURE 1).

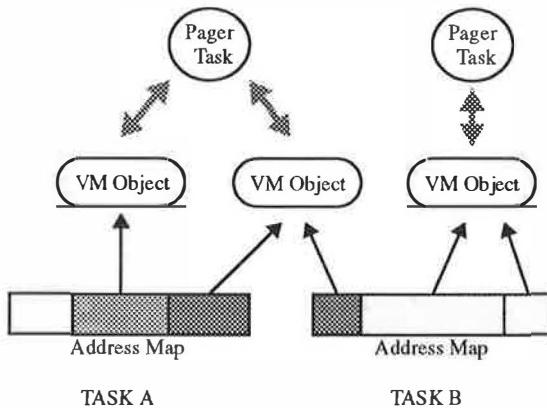


FIGURE 1 Mach VM System Structure

Beyond enabling a task to map a memory object into a specific part of its address space, the VM system also supports the shared access of multiple tasks to the same memory object (shared memory semantics) and the creation of lazy evaluated copies of a memory object (delayed copy semantics). While sharing of memory contents is easily accomplished by entering the same VM object into the address map of multiple tasks, delayed copy semantics are implemented by building shadow/copy relationships between VM objects.

Two different strategies are used for implementing delayed copy semantics, called symmetric and asymmetric copy strategy. With the symmetric copy strategy, the address maps of the source and the copy task continue to reference the same object. Only when a page is about to be written in either the source or the copy, a new object is created that has a shadow link to the original object. The new object replaces the original object in the address map where the write fault occurred (see FIGURE 2). While pages that are not present in the shadow object are retrieved from the source object through the shadow link, modifications of them take place in the shadow object. This means that the contents of the source object are frozen after a symmetric copy has been made.

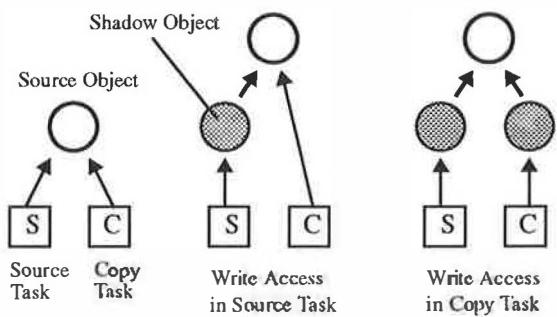


FIGURE 2 Symmetric Copy Strategy

While the symmetric copy strategy is a very efficient way of implementing delayed copy semantics, it is not applicable if changes in the source address space have to be reflected back to the source object's pager, such as with memory mapped files. Therefore, the asymmetric copy strategy is used in these cases. With this strategy, a copy object is created at the time a delayed copy operation is done. The source and the copy object are connected by copy and shadow links (see FIGURE 3). When a page is needed in the copy object, it is retrieved from the source object through the shadow link (pull operation). Before a page can be modified in the source object, first a copy of it is inserted into the copy object, using the copy link (push operation).

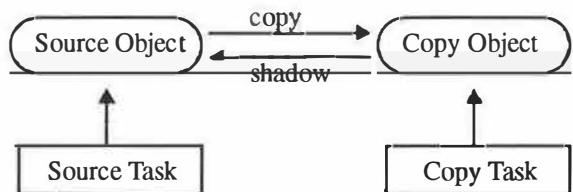


FIGURE 3 Asymmetric Copy Strategy

Multiple objects which are connected through copy links form a copy chain. New copies are inserted into the copy chain immediately after their source object.

An interesting property of both the symmetric and asymmetric copy strategies is that the creation of page copies is delayed until the page is modified in the source or copy address space. Pages that are retrieved through a shadow link in lieu of a read page-fault are not copied to the object where the fault occurred. Instead, the page-fault is satisfied by directly entering the source object page into the physical pagemap of the faulting task.

2.3 The eXtended Memory Manager

The original design of XMM is due to Joe Barrera, who created it in the context of a Ph.D. thesis at Carnegie Mellon University. Later, the Open Software Foundation (OSF) took over responsibility for further development of XMM [6]. In this section we describe the OSF NMK13 version of XMM, which is part of the software basis for the Paragon operating system and forms the basis for our performance comparisons. The major accomplishment in later OSF versions of XMM is a better integration of shared memory and delayed copy management, which closes a semantic gap in NMK13 XMM that prohibits combined use of shared and inherited memory.

2.3.1 Structure

XMM is located inside the Mach kernel and intercepts the communication between the VM system and external pagers. For each memory object XMM representations of the object are created on all nodes that make use of it. Only one of these representations holds state information and communicates with the pager task, while the others merely act as forwarding proxies for requests to and from their local VM system. This way XMM acts towards each node's VM system as the memory object's pager and towards the pager as a single node's VM system (see FIGURE 4).

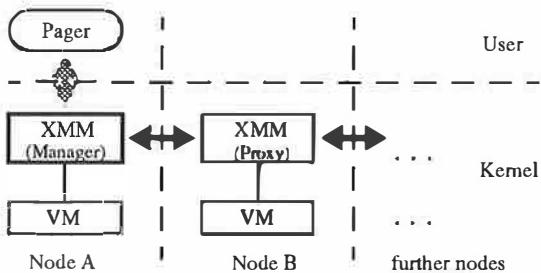


FIGURE 4 Global Structure of XMM

2.3.2 Shared Memory Management

XMM uses the centralized manager model for providing a coherent address space across multiple nodes. The manager node keeps track of the status of each page in the memory object's virtual address space on each node that makes use of the object and enforces a "single writer or multiple readers" policy. When some node makes a request for read or write access to a page of the memory object, this request is answered in two steps:

First, a coherent version of the page is created at the pager. This means that the page contents have to be returned to the pager if the page has been modified. Also, if the request is for write access to the page, the page has to be flushed from the VM cache of all nodes except the origin node of the request.

Then the request is forwarded to the pager, who can now view the requesting node as the only user of the page. The pager's answer is then forwarded to the requesting node.

2.3.3 Delayed Copy Management

NMK13 XMM supports delayed copy optimizations only in the context of remote task creation. The approach it takes toward the lazy evaluation of inherited memory is quite simple in that it leaves most of the work to the virtual memory system of the node where the source task is located.

This is done by creating a copy of the source address space just as in the case of a local fork() operation. Then a XMM internal pager is created for each memory object in the copy address space, thereby providing a new memory object abstraction to be mapped in the address space of the remote task. Page faults in the remote task's address space cause a request to the XMM internal pager which in turn generates a page-fault on the local copy address space and supplies the resulting page contents back to the remote node.

3. The ASVM system

3.1 Design Principles

Based on our experiences with XMM, we developed our new system along the following guidelines:

- **Distributed Manager**

XMM is based on a centralized manager approach for managing cross-node memory relationships. This means that for each Mach memory object, the XMM code on a single node is responsible for coherency and copy management. Several research studies ([1], [10]) indicate that this approach is non-scalable and thus unsuitable for management of shared virtual memory on large systems, because the centralized manager becomes a bottleneck when a large number of nodes uses the memory object.

ASVM uses a distributed manager approach where each page has its own manager, which is also called the page owner. The ownership can migrate between all nodes that use the page.

- **Limited Memory Requirements**

With XMM, the centralized manager stores the page state of a memory object in a data structure that requires 1 byte of non-pageable memory for each page in the virtual address space of the memory object, multiplied by the number of nodes that use the object. With large numbers of nodes or large and sparsely populated address spaces this concept can consume a lot of memory. In extreme cases it may even consume more memory than is actually available, leading to a system crash.

ASVM not only distributes the page state information across the system, but also ties it to physical pages in that a node only holds state information about pages that are cached into its physical memory. The same concept is used by the Mach kernel's virtual memory system to support large and sparsely populated address spaces.

- **Asynchronous State Transitions**

One problem of the mechanism XMM uses for implementing its delayed copy support is that the copy pager thread which generates a page-fault is blocked until the page-fault completes. As an inter-node copy chain might cross the same node multiple times, this leads to a deadlock if the available number of threads is exhausted.

To avoid problems of this kind, ASVM generally uses asynchronous state transitions, which means that neither a thread nor other resources are blocked while waiting for a request to be answered.

- **Specialized Communication Protocol**

XMM uses a protocol called XM^IM (eXtended Memory Management Interface) for communication inside the XMM system. XM^IM is an extension of the EMMI protocol that defines the interaction between the virtual memory system and an external pager [4]. Although this is an elegant solution, it is also very inefficient when used for coherency management, as it requires more messages than necessary. For example, transferring a write permission from one node to another using XM^IM takes five messages, two of them containing page contents. With a more suitable protocol, this number could be reduced to three messages (request to manager, request from manager to cur-

rent writer, and answer from current writer to new writer), only one of them containing page contents [7].

ASVM uses XM^IM only as an interface to the local VM system and pager while defining its own ASVM protocol for all communication between the ASVM instances on different nodes.

- **Dedicated Transport Service**

XMM uses Mach NORMA IPC as a transport service for XM^IM communication to remote nodes. This introduces high latencies and overhead due to the handling of port rights and complex message structures. This is especially conspicuous on multi-processor systems with a high-performance interconnect such as the mesh used in Paragon systems. In fact, on these systems NORMA IPC is responsible for about 90 percent of the latency involved in resolving remote page faults for memory that is shared through XMM.

The ASVM protocol, on the other hand, is mapped to a dedicated transport interface, the SVM transport service (STS). This allows the use of lower level protocol stacks for ASVM communication that can take advantage of the simple structure of ASVM messages, which consist of a fixed size block of untyped data (currently 32 Byte), possibly followed by the contents of a VM page (8 KByte). Also, flow-control is simplified by the fact that page contents are only transferred on behalf of a request from their receiver, which allows preallocation of page receive buffers.

3.2 Basic Operation

For managing coherency and memory inheritance, some entity must maintain information about the current state of a page. This entity is called the manager of the page. The main design freedom for any SVM system is the method used to distribute the managers of particular pages across the nodes of the system. There are several known approaches to this problem [1]:

- **Centralized Manager**

The managers for all pages of a memory object are located on the same node. This approach is implemented in NMK13 XMM.

- **Fixed Distributed Manager**

The page managers are statically distributed by using some function which maps a page number to a node number. The manager for the page is located on that node.

- Dynamic Distributed Manager

The manager of a page migrates between the nodes that use the page. The node on which the manager of a page is currently located is called the owner of the page. Kai Li [1] suggests moving page ownership to the node that most recently made a request for the page and locating the actual page owner by following a *hint chain* across all the nodes that previously owned the page. This hint chain naturally collapses when a node that once owned a page becomes its owner again. The hint chain can also be collapsed while forwarding a request as the originator of the request becomes the next owner.

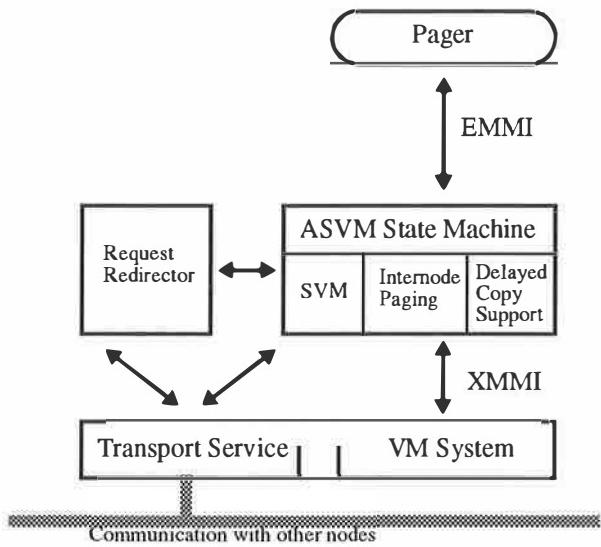


FIGURE 5 Structure of the ASVM System

ASVM adopts the dynamic distributed manager approach, but makes a clearer distinction between the method for distributing page managers and the method for finding the current manager of a page. In this approach, there are actually two managers for a page. One of them is the page owner, which keeps information about the state of the page and is responsible for answering access requests from other nodes. The other is the ownership manager, which is responsible for forwarding requests to the page owner. While a page is always owned by the node that most recently had write access to it, multiple methods are available for forwarding requests to the page owner. This is different from Kai Li's approach, in which page ownership also migrates on read requests and a single method is used for forwarding requests (the hint chain).

3.3 General Structure

FIGURE 5 shows the structure of the ASVM system and how it is embedded into the Mach kernel.

When the virtual memory system of a node needs access to some page, an access request is generated and fed into the request redirector, which is responsible for delivering it to the current page owner. When the request arrives on the page owner node, it is fed into a state machine which keeps the page coherent throughout shared usage, delayed copying, and internode paging.

3.4 Forwarding Mechanisms

The motivation for providing multiple strategies for finding the owner of a page comes from the design goal of avoiding data structures that grow with the size of virtual address spaces. This goal is guaranteed for the page management structures by enforcing the invariant that each node that is owner of a page has the page in its virtual memory cache. If no node has the page in its VM cache, there is no owner of the page and no state information about it. The pager can be viewed as the page owner in this case.

A different approach must be taken for page ownership information, since each node must also hold ownership information for pages that aren't in its virtual memory cache. Consider the following forwarding strategies:

- Dynamic forwarding
Requests for a page are forwarded as in Kai Li's dynamic distributed manager approach.
- Static forwarding
Requests for a page are forwarded through a fixed distributed ownership manager.
- Global forwarding
Requests for a page are forwarded through all nodes until they eventually reach the page owner or - if there is no owner of the page - the pager.

Dynamic forwarding requires the most memory since each node has to hold an ownership hint for each page in the virtual address space. Static forwarding uses less memory since each node only holds ownership information for a subset of the pages in the virtual address space. Global forwarding uses the least memory since only the page owner itself has to know that it owns the page.

ASVM implements all three of these strategies and holds ownership information for dynamic and static forwarding in caches for the most recently accessed pages (see FIGURE 6). This means that both dynamic and static forwarding can fail if required information is not present in the caches. Static forwarding will not fail as often as dynamic forwarding since the static forwarding cache is in effect distributed among all static ownership managers and thus can hold many more owner references than a dynamic cache of same size. Therefore, static forwarding is used as a backup for dynamic forwarding and is in turn backed up by global forwarding. Global forwarding will never fail if any node is owner of the page.

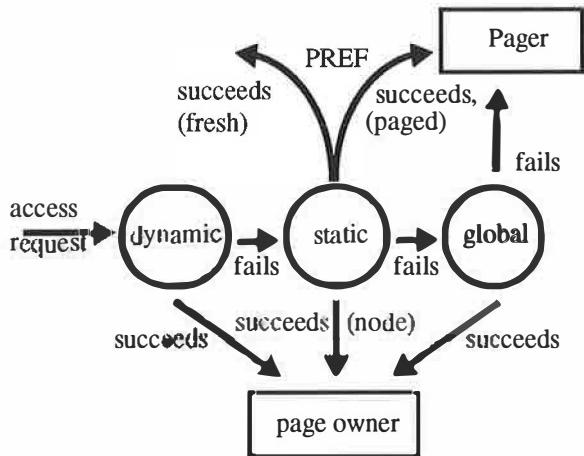


FIGURE 6 Forwarding Mechanisms

In addition to a node reference the static cache can also hold hints that the page has never been initialized (fresh) and that the page has been paged out (paged), thereby avoiding costly global forwarding operations in these cases.

The ASVM system allows to disable either dynamic or static forwarding (or both) on a memory-object basis. This provides great flexibility. If only static and global forwarding are enabled, the behavior of the ASVM system is identical to Kai Li's fixed distributed manager approach. Enabling dynamic forwarding makes the ASVM system resemble the dynamic manager approach. Of course this assumes that in both cases the relevant caches are big enough.

3.5 Shared Memory Management

The only coherency model that is currently supported by ASVM is *strong coherence*, which means that any read operation to a shared memory address will return

the data of the most recent write operation to this address.

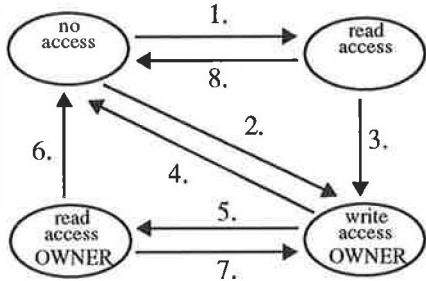


FIGURE 7 SVM state diagram

FIGURE 7 shows the part of the ASVM state machine that keeps a page coherent through shared usage, enforcing the single writer or multiple readers invariant. The state of a page on a particular node includes the type of access the node's VM system has to the page and an indicator if the node is owner of the page. The following state transitions can occur:

1. The node is granted read access to the page.
2. The node is granted write access to the page.
3. The node is granted an upgrade from read to write access.
4. The node is owner of the page and grants write access to another node.
5. The node is owner of the page and grants read access to another node. This node is also entered into a list of nodes with read access (reader list)
6. The node is owner of the page and grants write access to another node. An invalidation message is sent to all nodes in the reader list.
7. The node is owner of the page and upgrades its own access rights from read to write access. An invalidation message is sent to all nodes in the reader list.
8. The node receives an invalidation message from the page owner.

3.6 Internode Paging

The main idea behind the concept of internode paging is to extend not only the semantics of the VM system across node boundaries, but also its concept of managing physical memory as a cache for virtual memory contents. Together with the mechanisms for maintaining coherency, ASVM's internode paging facilities allow it to view the physical memory of all nodes that use a particular memory object as a cache for the memory object's contents.

When a page is evicted from a node's virtual memory cache, the following algorithm is used:

1. If the node is not the page owner, the page is simply discarded, as it can be retrieved from the page owner at any time.
2. If the node is the page owner and its list of nodes with read access is not empty, these nodes are asked, one after another, if they still have read access to the page (they might have discarded the page as in Step 1). If a queried node answers to have no read access to the page, the node is removed from the reader list. Else ownership for the page is transferred to this node. Note that this ownership transfer doesn't require sending the page contents.
3. If the node is the page owner and there are no more nodes with read access to the page, ASVM tries to transfer the page to one of the other nodes that have mapped the memory object to which the page belongs. This fails if no node with sufficient free memory can be found.
4. Finally, if Step 3 failed the page is returned to the memory object's pager.

For selecting a pageout node in step 3, a counter is used that cycles through the list of nodes which have mapped the memory object. This counter is incremented on each pageout. First, the node identified by the current counter value is asked to accept the page transfer. If this node doesn't accept the page transfer because it is low on memory, the node which most recently accepted a page transfer is asked again.

This algorithm is meant to adapt itself to the current memory situation by locking onto nodes which are known to have free memory available until another node is found which also has memory available. If many nodes have free memory, such as when a single node initializes a SVM region that is mapped on multiple nodes, the pages are distributed evenly among the other nodes. This provides good load balancing for ASVM's distributed manager algorithms.

3.7 Delayed Copy Management

ASVM extends the asymmetric copy strategy of the virtual memory system across node boundaries. The basis for internode copy relationships is provided by the shared virtual memory functionality of ASVM. When a copy of a memory object is to be mapped on some node, first a shared mapping of the source object is established on this node. Then a local copy is created through the standard mechanisms of the VM system (see FIGURE 8). After that all resident pages of the source object on all nodes which share the object are marked read only by sending a message to the

sharing nodes which causes them to do a `memory_object_lock_request` on the copied region.

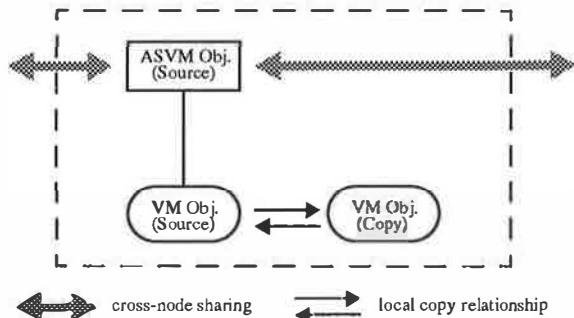


FIGURE 8 Delayed Copy Layout

While all local push and pull operations continue to be done internally in the VM system, ASVM has to be involved if either the source or the copy object are shared between multiple nodes:

- If the source object is shared, a page which is about to be modified has not only to be pushed to the local copy object, but also to copy objects on all other nodes which share the source object.
- If the copy object is shared, before pushing a page to the copy object it has to be determined if the page is already present on any of the nodes which share the copy object.

3.7.1 Extensions to the EMMI Interface

The shared virtual memory functionality of ASVM was implemented without changing the EMMI. This was not possible with the delayed copy management, because the EMMI design hides the existence of copies and shadows from the memory manager. So the following EMMI extensions had to be made to give the memory manager control over the VM internal copy mechanisms:

`memory_object_lock_request` was extended by a "mode" argument that allows to specify if the page should be pushed down the VM internal copy chain before executing the lock operation.

`memory_object_lock_completed` was extended by a "result" argument which is used to return an indication if the lock_request couldn't execute a push operation because the page was not present in the VM cache.

`memory_object_data_supply` was extended by a "mode" argument that allows to push a page down the copy chain instead of supplying it to the source object.

memory_object_pull_request was added to allow a page to be retrieved through the VM internal shadow chain.

memory_object_pull_completed was added to return the result of a pull_request. There are three possible results:

1. The page is not available and can be zero-filled.
2. The page is available and its contents are returned.
3. The memory manager of a shadow object has to be asked for the page and the shadow object port is returned.

3.7.2 Push operations

A page has to be pushed into all (possibly remote) copies of a memory object before it can be modified in the source object. A push operation is initiated by the current owner of a page if he receives a write request and the page has not already been pushed into all copy objects.

To determine if a page needs a push operation, ASVM uses version counters for memory objects and pages: An object's version counter is incremented each time a copy is made from the object. The version counter of a page is set to the associated object's version counter each time a push operation takes place on the page. If a page is about to be modified and the page version is not equal to the associated object's version, a push operation is initiated before write access is granted.

When a push operation takes place for a page, a message is sent to all nodes who have mapped the associated memory object, except the node who initiates the push operation. On these nodes the **memory_object_lock_request** EMMI call is used for prompting the VM system to push the page down the copy chain and invalidate it in the source object. If the page is not present in the VM cache, the reply to the lock request will indicate this and a request to send the page contents is sent to the page owner. Once the request is answered, a **memory_object_data_supply** EMMI call is made to push the page down the copy chain.

Once the page owner received replies from all nodes which were requested to push the page and after all missing pages have been sent to these nodes, the source object side of the push operation is completed. Additional effort is required on the side of the copy objects if they are shared. To determine, if a page is already present in a shared copy object, a special type of access request, called a push scan request, is generated and forwarded through ASVM's forwarding

mechanisms. If a page owner exists in the copy object, it will answer the push scan request and the push operation will be canceled for this copy object. If no page owner exists in the copy object, the request will finally be forwarded to the shadow object and the push operation will proceed for this copy object.

3.7.3 Pull operations

For pull operations, ASVM uses the VM system to traverse local shadow chains and its own SVM capabilities to bridge the gap between node boundaries. If a page-fault occurs in a VM object, the VM system will traverse the local shadow chain and - if the page isn't present in one of the traversed source objects - generate a **memory_object_data_request** EMMI call in the first object that has an associated ASVM object. ASVM will then forward this request through its forwarding mechanisms. If the page is present on some node, it will have an owner who receives and answers the request.

Else - assuming that the current object is a copy too - the request will finally be forwarded to the node on which the copy was created and that also has a mapping of the corresponding source object. On this node, which is also called the peer node of the current object, ASVM will then use the **memory_object_pull_request** EMMI call to traverse the local shadow chain. If the requested page is present in one of the source objects on this node or can be zero-filled, it will be supplied to the origin node of the request.

If the result of the **pull_request** indicates that the page might be present in a source object which has an associated ASVM object, the request will be forwarded into this object. This continues until either the page is found in some object or the end of the shadow chain is reached, in which case the page can be zero-filled.

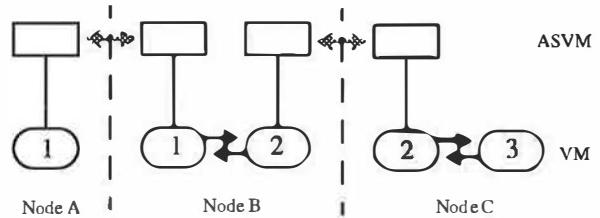


FIGURE 9 Copy chain across multiple nodes

For an example, FIGURE 9 shows a copy chain across two nodes as it is created if a task forks to a remote node and the child task does the same. Assume that a page-fault occurs in object 3 on Node C and the page is located in object 1 on Node A. The VM system on

Node C issues a `data_request` for the page in object 2. ASVM forwards the request to Node B, which is the peer node of object 2, and uses a `pull_request` to traverse the local shadow chain. The result of the `pull_request` indicates that the page has to be looked up in object 1 and ASVM forwards the request to Node A. Here, again a `pull_request` is used and returns the page contents. ASVM then supplies the page to the object from which it got the request, object 2 on Node C. Finally, the VM system will take care of entering the page into the physical pagemap of the task that took the page-fault in object 3.

One issue left out so far is that of synchronization between push and pull operations. Because of the distributed nature of ASVM's page management, a request from a copy object can enter its source object at the same time a push operation is in progress. In this case, the copy request is held up until the push operation completes. Then a retry indicator is set in the request and it is sent back to the origin node, which causes the request to be repeated.

4. Performance Measurements

We made our performance measurements on a Paragon multicomputer system with 72 GP nodes (2 processors and 16 MByte memory per node).

4.1 Basic Page-Fault latencies

4.1.1 Page-Faults on Shared Memory

Table 1 compares characteristic types of SVM page faults and their latency under the ASVM system and NMK13 XMM. All latencies were measured in user-task context by performing read or write operations and timing their duration (in milliseconds).

FIGURE 10 is a graphical representation of the latency introduced by a write page fault in relation to the number of nodes that have read copies of the page. The figure distinguishes between two cases. In one, called *write upgrade fault*, the faulting node already has a read copy of the page. In the other, called *write fault*, the faulting node doesn't have a read copy. The NMK13 XMM times are measured for the general case in which the XMM stack is remote from both the faulting node and the nodes that have read copies.

The graph shows that the page fault latencies of ASVM increase much slower with the number of nodes that have a read copy of the page than the page

Table 1: Page Fault Latencies

Fault Type	ASVM	XMM
Write fault on a page with 1 read copy	2.24	38.42
Write fault on a page with 2 read copies	3.10	12.92
Write fault on a page with 64 read copies	8.96	72.18
Write fault on a page with 2 read copies, faulting node has read copy	1.51	3.83
Write fault on a page with 64 read copies, faulting node has read copy	7.75	63.72
Read fault on a page, faulting node is first reader	2.35	38.59
Read fault on a page, faulting node is second reader	2.35	10.06

fault latencies of NMK13 XMM. This is especially important for SVM applications and is one indicator for the better scalability of ASVM.

The huge difference between page faults with only one read copy and with two read copies that was measured for NMK13 XMM can be explained by the fact that XMM writes a dirty page to the paging space when it is requested for the first time by another node.

4.1.2 Page-Faults on Inherited Memory

For evaluating the performance of delayed copy operations across node boundaries we used a test program that initializes a region of memory (128 KByte), spawns a chain of copies of that region across a defined number of nodes and faults in all pages of the region on the last node in the copy chain. FIGURE 11 shows the resulting page fault latencies both for NMK-13 XMM and ASVM. They can be described as $lb + n * la$, where lb is the basic latency for a remote copy-on-access fault (5.0 ms for NMK-13 XMM and 2.7 ms for ASVM) and la is the cost for forwarding the page fault across an additional node (about 4.3 ms for NMK-13 XMM and about 0.48 ms for ASVM).

The low additional costs ASVM incurs on faults that have to be forwarded across a long copy chain is espe-

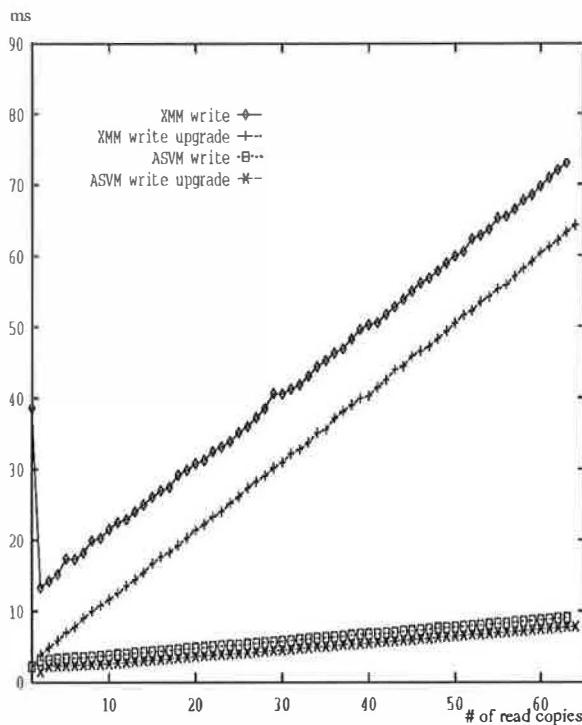


FIGURE 10 Write Fault Latencies for a Page with N Read Copies

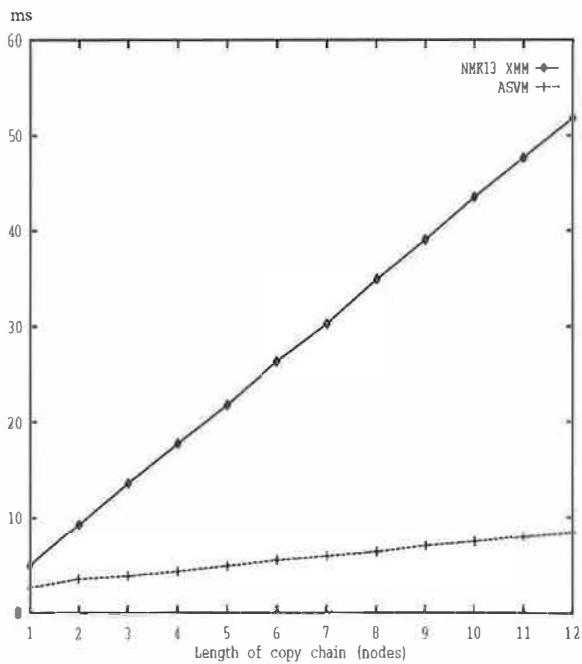


FIGURE 11 Latency for page faults across a copy chain

cially important for applications that use dynamic load balancing, as each migration of a task adds another stage to the copy chain from the node where the task

has originally been started to the node where it is running. But also normal parallel applications take a considerable advantage: The Paragon operating system spawns applications to a set of nodes by forking along a binary tree, which creates copy chains of a maximum length that grows logarithmically with the number of nodes involved. For example an application which is started on 256 nodes creates copy chains with a maximum length of 8. A page fault across a copy chain of length 8 is associated with a latency of 35 ms for NMK-13 XMM and 6.4 ms for ASVM.

4.2 Mapped Filesystem Performance

While the Paragon OS uses a memory mapped unix file system, parallel access to the same file is sequentialized by the OSF1/AD TNC server, which lacks multiple reader semantics. Therefore, the measurements in this section don't use standard Unix read/write calls. Instead, they bypass the server by using the mmap() function to map a file into memory and reading/writing directly from/to memory. The performance of standard read/write calls can be expected to come close to the values presented here once multiple reader semantics are added to the OSF1/AD server. TABLE 2, FIGURE 13 and FIGURE 12 show the effective transfer rates seen by each of multiple nodes accessing the same file.

The write transfer rate is measured by letting all nodes write different sections of a 4-MB file. Asynchronous writes were used, so the upper limit for the combined transfer rate of all nodes is given by the rate at which the file pager can supply initially zero-filled pages for the file. The read transfer rate is measured by letting all nodes read a 4-MB file in parallel. The upper limit for the individual transfer rate of each node is given by the rate at which the file pager can supply the contents of the file.

TABLE 2 File Transfer Rates (MB/s)

	1	2	4	8	16	32	64
ASVM write	2.80	2.60	2.05	1.22	0.62	0.30	0.15
XMM write	2.15	1.77	0.90	0.49	0.24	0.12	0.06
ASVM read	1.57	1.53	1.14	0.91	0.70	0.66	0.66
XMM read	1.18	0.38	0.25	0.11	0.05	0.02	0.01

Note that because of its distributed manager concept, ASVM is able to sustain a reasonable read transfer rate even on a high number of nodes. To achieve the same

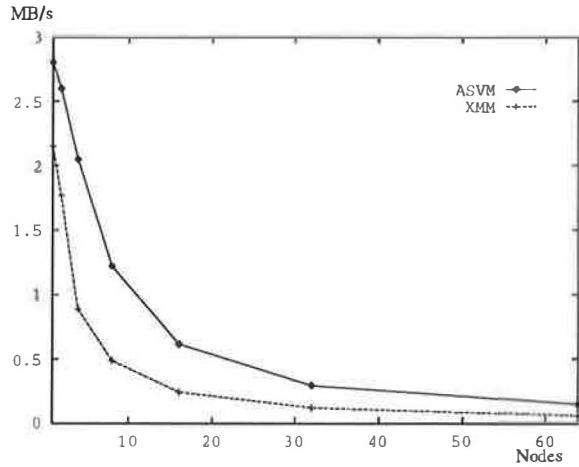


FIGURE 12 Parallel Write Bandwidth for Memory Mapped Files

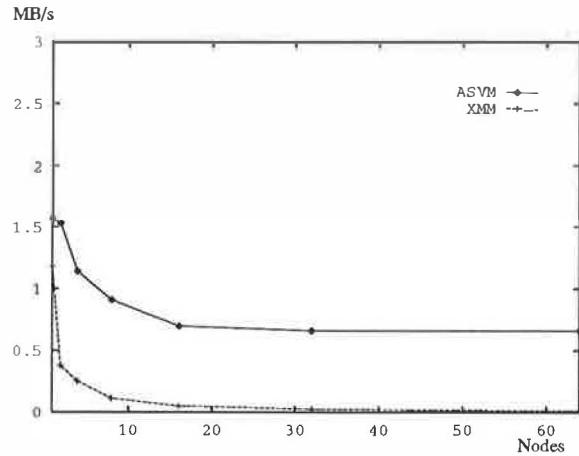


FIGURE 13 Parallel Read Bandwidth for Memory Mapped Files

for write transfers, the file pager would have to tell the kernel about chunks of uninitialized data, so that ASVM's distributed mechanisms could be used for supplying initially zero-filled pages.

4.3 SVM Application Performance

This section presents performance results based on a program called EM3D. This program's communication was originally based on active messages [9], which we changed towards shared memory based communication. EM3D models the three dimensional propagation of electromagnetic waves. Its basic data structure is a bipartite graph that has directed edges from a set of E nodes representing the electric field to a

set of H nodes representing the magnetic field, and vice-versa. After an initialization phase for building up the graph, the computation consists of a large number of iterations to calculate the development of the fields over time. In each iteration the value of each E node is changed to the weighted sum of the H nodes to which it is connected, and then the value of each H node is changed to the weighted sum of the E nodes to which it is connected. To prevent confusion, the E and H nodes of the EM3D code will be called cells.

The graph for the performance measurements was generated randomly with a user-specified percentage (20%) of the edges (6 per cell) leading to a cell located on a different processing node. The execution times are given in seconds for 100 iterations of the computation loop. The initialization phase wasn't included in the measurements since practical applications would use even more iterations, making the initialization overhead almost vanish.

TABLE 3 demonstrates the effect of ASVM's enhanced scalability on SVM applications and shows the execution times of EM3D for various problem sizes both for NMK13 XMM and ASVM. The column headings indicate the number of nodes on which the application is run while the row headings indicate the use of XMM or ASVM and the problem size, given by the number of cells. With ASVM the execution times decrease with the number of nodes, resulting in reasonable speedups while with NMK13 XMM the execution times actually increase, resulting in a slowdown.

TABLE 3 EM3D Timings^a (seconds)

EM3D	1	2	4	8	16	32	64
ASVM 64000	43.6	32.0	19.9	13.9	11.2	9.86	9.55
XMM 64000	43.6	151	213	392	755	1405	2735
ASVM 256000	174*	**	**	33.6	21.5	15.6	12.8
XMM 256000	174*	**	**	520	842	1604	2957
ASVM 1024000	698*	**	**	**	**	54.2	24.4
XMM 1024000	698*	**	**	**	**	1863	3373

- a. * estimated timings without paging.
- ** meaningless because of paging effects.

Note that each cell in the problem space needs 224 bytes of memory. This means that 64,000 cells consume 14 MB, which is already too much for a 16-MB processing node that only has about 9 MB of memory available for user applications. The sequential measurement with 64,000 cells was therefore made on a 32-MB node. The parallel measurements were omitted where the combined memory of the 16-MB compute nodes wasn't sufficient to hold the complete data set.

5. Conclusions

Our first evaluations indicate that ASVM fully achieves its goal of providing efficient and scalable distributed memory management. The main reason for this success lies in the close interaction between the three ASVM subsystems for shared virtual memory management, delayed copy support and internode paging. To demonstrate this interaction, we will take a closer look at how it helps in achieving scalability and efficiency:

- Scalability

As described in the section 3.5, the owner of a page keeps a list of nodes with read access to the page. The maximum size of this list grows linear with the number of nodes in the system, which poses a problem for scalability. This is where ASVM's internode pageout mechanisms enter the game: The greater the number of nodes with read access to a page, the greater the number of nodes which are available for an ownership transfer when memory gets scarce and the page is evicted from the VM cache. Such, ASVM's internode pageout mechanisms effectively balance the amount of owner information each node has to keep among all nodes which share a memory region.

- Efficient Memory Usage

A great part of the memory inherited to a child process is not modified by either the parent or the child. The most prominent example for this are program text segments. As ASVM uses the VM system for establishing local copy relationships, pages requested through a read page fault will be supplied to the source object on the requesting node, instead of the copy object (see 2.2). This means that paging for read-only pages of copy objects is done through their source object, taking full advantage of ASVM's internode pageout mechanisms. If, for example, a parallel application

is loaded onto a number of nodes and memory gets scarce, ownership for all read-only pages will be distributed among the participating nodes and not owned pages are discarded. If a node later needs access to a page it has discarded, the page can be retrieved from its current owner instead of reading it from a disk.

6. Future Work

While ASVM provides a solid foundation for establishing an efficient and scalable single system image, a number of modifications has to be made to the rest of the Paragon OS to utilize ASVM's full potential. We will concentrate on the area of file systems here.

Currently, the Paragon OS provides two types of local file systems, the Unix File System (UFS) and the Parallel File System (PFS). The Unix File System is a memory mapped filesystem and thereby uses the VM system to provide data buffering directly on the nodes which use a file. On the other hand, the Parallel File System supports striping of files across multiple I/O nodes and uses NORMA-IPC to distribute read/write requests to the I/O nodes. The main advantage of UFS is its caching scheme that becomes even more efficient through ASVM's internode paging facilities, while the main advantage of PFS is its scalability through the use of multiple I/O nodes.

In the following we will hint at what is necessary to combine the advantages of UFS and PFS into a new filesystem that supports striping, local caching and full Unix file semantics without a loss of performance:

- Provide and utilize ASVM primitives for locking a range of pages in a shared address space for the exclusive access of a particular task on a particular node. This would allow to guarantee the atomicity of read and write operations by locking the range which is about to be modified prior to write operations. The current scheme uses NORMA-IPC to acquire an exclusive token from a token server each time a read or write operation takes place and the token is not present on the node which does the file access.
- Modify the VM system to allow multiple pagers for one VM object that are used for paging requests in a round-robin fashion. This allows one VM object to represent a striped file system, with one pager located on each of the I/O nodes.

- A clustering of page-out and page-in requests has to be implemented in the virtual memory system and in ASVM to achieve adequate bandwidths.

Acknowledgments

This research was sponsored by the Advanced Research Projects Agency under contract MDA972-89-C-0034. It has also benefitted from discussions, suggestions and feedback by many people, including our colleagues at Intel's Scalable Systems Division, Fritz Gerneth - Intel GmbH and Michael Gerndt - KFA Juelich.

References

- [1] Kai Li: **Shared Virtual Memory on Loosely Coupled Multiprocessors**. Ph.D. thesis, Yale University, September 1986
- [2] Roman Zajcew, Paul Roy, David Black et al: **An OSF/1 UNIX for Massively Parallel Multicomputers**. In Proceedings of the USENIX conference, January 1993.
- [3] Richard F. Rashid: **Threads of a new system**. Unix Review Vol 4(8), August 1986
- [4] Michael Wayne Young: **Exporting a User Interface to Memory Management from a Communication-Oriented Operating System**. Ph.D. thesis, Carnegie Mellon University, November 1989
- [5] Avadis Tevanian, Jr.: **Architecture-Independent Virtual Memory Management for Parallel and Distributed Environments**. Ph.D. thesis, Carnegie Mellon University, December 1987
- [6] Bill Bryant, Steve Sears, David Black, Alan Langerman: **An Introduction to Mach 3.0's XMM Subsystem**. OSF Research Institute Operating Systems Collected Papers Vol. 2, October 1993
- [7] Alessandro Forin, Joseph Barrera, Michael Young, Richard Rashid: **Design, Implementation and Performance Evaluation of a Distributed Shared Memory Server for Mach**. Report CMU-CS-88-165, Carnegie Mellon University, August 1988
- [8] Pete Keleher, Sandhya Dwarkadas, Alan L. Cox, Willy Zwaenepoel: **Treadmarks: Distributed shared memory on standard workstations and operating systems**. In Proceedings of the USENIX conference, January 1994
- [9] Culler, D.E., Dusseau, A., Goldstein, S.C., et al.: **Parallel Programming in Split-C**. In Proceedings of the Supercomputing 93, 1993
- [10] Lahjomri, Z., Priol, T.: **Koan: A Shared Virtual Memory for the iPSC/2 Hypercube**. In Proceedings of the CONPAR conference, 1992

The Authors

Stephan Zeisset currently works as an International Project Leader at Intel's Scalable Systems Division. He graduated in 1994 with a Masters degree in Computer Science from the Munich University of Technology. His masterthesis built the basis of the XMM rewrite project, in which he took part as a Software Engineer at Intel GmbH. His e-mail address is sz@ssd.intel.com.

Stefan Tritscher is a Technical Marketing Engineer at Intel GmbH. Prior to joining the marketing department he worked as a Senior Software Engineer on Paragon OS components such as load balancing and the XMM rewrite project. He is holding a Masters degree in Computer Science from the Munich University of Technology. His e-mail address is stefan@esdc.intel.com.

Martin Mairandres is currently pursuing a Ph.D. degree from RWTH Aachen, Germany. He is also holding a Masters degree in Computer Science from the Munich University of Technology. During his Ph.D. research at Intel GmbH he contributed a lot to the shared virtual memory functionality of ASVM and designed interfaces for system and application level monitoring. His e-mail address is martinX@esdc.intel.com.

Trademarks

Paragon™ is a trademark of Intel Corporation.

Fault Tolerance in a Distributed CHORUS/MiX System

Sunil Kittur[†]

Douglas Steel

ICL High Performance Systems, Manchester, UK

François Armand

Jim Lipkis

Chorus Systems, Saint-Quentin-En-Yvelines, France

ABSTRACT

Within a distributed system, resources may be shared between nodes. The system should continue to operate even if individual nodes fail due to hardware or software errors. This may result in the loss of resources that were hosted on the failed node, but it may be possible to continue to provide access to some resources by hosting them on another node.

This paper describes mechanisms that allow the failover of resources from failed nodes. Failover is currently restricted to disk volumes and file systems. The failover mechanisms maintain the correct semantics at the UNIX system call level for operations from surviving nodes that were in progress at the time of the failure, including non-idempotent operations.

Minimal resource and performance overheads are imposed for the normal running case, and in contrast to replication techniques, state is recovered and rebuilt at the time of a failover.

1. Introduction

The GOLDRUSH system, developed at ICL, is a distributed memory multi-computer consisting of up to 64 nodes connected by a high-speed interconnect. Each node contains up to 12 SCSI disks that are only physically accessible by that node, and some nodes contain FDDI couplers which provide external access to the machine (Figure 1).

Each node is a self-contained UNIX system running a version of the CHORUS/MiX V.4 operating system, with some devices and file systems accessed from remote nodes using CHORUS IPC protocols [Rozier88, Batlivala92]. CHORUS/MiX CHORUS/MiX implements SVR4 UNIX as a

“personality” on top of the CHORUS microkernel. It consists of a set of independent actors (a process-like abstraction) which run in supervisor mode. These actors include, the Process Manager (PM) which receives system calls from UNIX processes and acts as a client on behalf of these processes; and the Object Manager (OM) which provides file and device access.

The main application of the machine is to provide a parallel database server, and a number of commercial databases such as Oracle, Ingres and Informix have been ported to it. Administrative software is used to provide a consistent view of shared operating system resources across nodes.

To provide a highly available database service, the system must continue to function in the event of the failure of disks or nodes.

To provide resilience to individual disk failures, disk volumes are mirrored, so that if one of the mirrors fails, the volume is still available from the remaining mirror. The mirror may be a remote device; this functionality is provided by the Distributed Plex Manager (DPM) as shown in Figure 2. As mirroring techniques are fairly standard, the rest of this paper will discuss the handling of node failures.

Node failure can cause the loss of applications and disks/file systems that were hosted on that node. In the case of software errors, such as a kernel panic, it may be possible to reboot the node, and resume communication with the node, possibly performing some recovery actions. This is the approach used in Sprite [Baker94], NFS [Sandberg85] and Spritely NFS [Mogul92]. However, this can result in a considerable delay, since a reboot involves much more than the recovery of just the shared devices and file systems. This approach will also fail if the node cannot be rebooted, for example, if there was some permanent hardware failure.

[†] Author's current affiliation is at Online Media, Cambridge, UK.

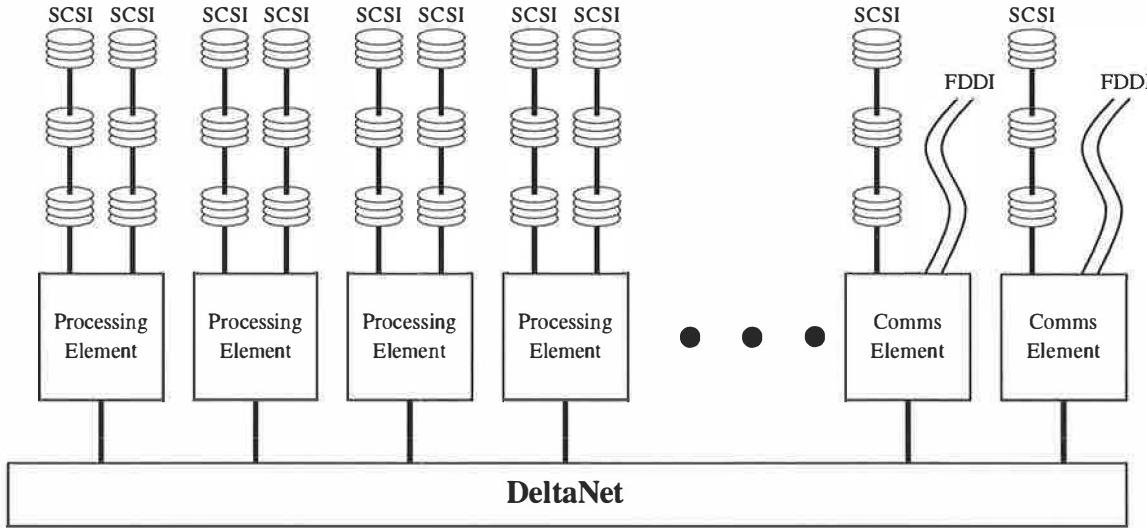


Figure 1. GOLDRUSH system architecture

Instead, we make the device or file system available from another node by a failover mechanism which allows access to the resource as soon as the appropriate recovery actions have been completed (Figure 3). This is similar in some respects to the HA_NFS work [Bhide91], but our approach covers more than just the NFS protocol.

It is important that this failover be transparent to the users of the resource, and in particular, the expected semantics of any operations that were in progress at the time of the failure must be preserved. This is not a problem for idempotent operations, since they can simply be retried once the failover has completed. However, non-idempotent operations, such as `unlink(2)` or `write(2)` on a file opened in `O_APPEND` mode should be retried only if the operation had not completed before the failure occurred.

2. Goals and Tradeoffs

We identified a number of key goals for GOLDRUSH:

- There should be no single point of failure that causes the entire system to be shut down.
- Applications not using resources on a failed node should continue without disruption.
- Performance and resource overheads should be minimised.
- The performance of normal operation is more important than the speed of recovery.
- The semantics of all UNIX APIs should be preserved during and after recovery.

- If a resource cannot be recovered, it should be cleanly removed, including cleaning up any state on surviving nodes.

Whilst satisfying these goals, our initial implementation makes a number of tradeoffs:

- The system does not tolerate double failures; if a second node fails before the first failure is recovered, the entire system may be shut down.
- The mechanisms are intended to cover only operating system resources; applications are responsible for handling the failure of application components that were on failed nodes.
- There is no need for instantaneous recovery; some short delay is acceptable. Providing instantaneous failover would require some form of replication, which would add considerable overheads to the normal running situation.
- The initial implementation covers disk volumes and file systems, but the mechanisms should be flexible enough to accommodate other types of resource.
- Data to remote file systems is written data-synchronously. This avoids having to recover from the loss of nodes containing dirty cached data. Synchronous writes are ideal in any case for database servers, which perform their own caching and have no reason to incur the cost of disk block copying required for buffered I/O. However, if general time sharing had been a goal, there would be a performance hit.

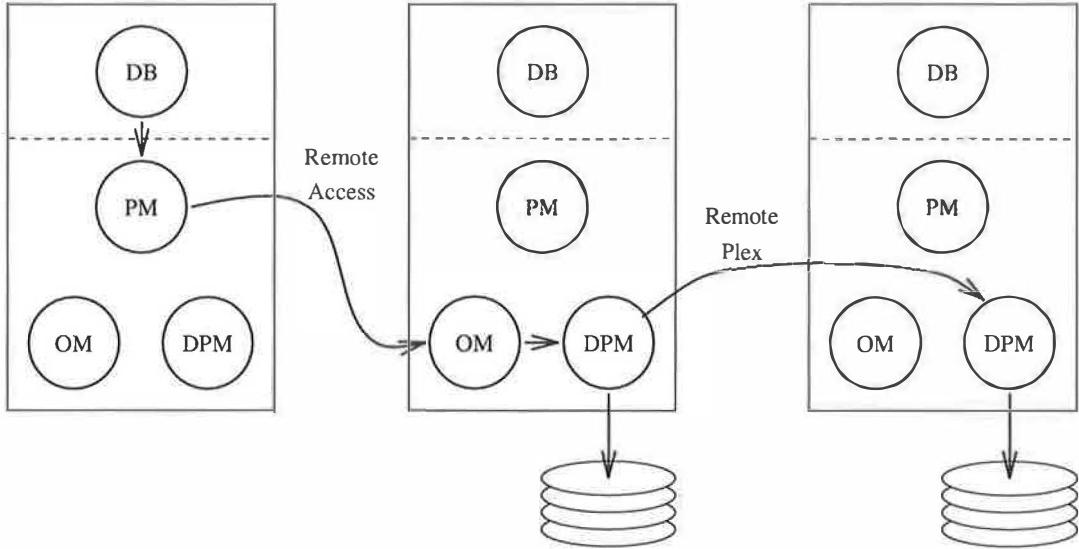


Figure 2. Access to a remote volume/filesystem

3. Architecture

The GOLDRUSH system can be viewed as a collection of homogeneous UNIX machines connected by a high-speed private network. The memory and disks on each node are only physically accessible by that node. Remote access is provided by software using CHORUS IPC. This is feasible because of the high bandwidth available.

Resilience to disk failure is provided using the VERITAS VxVM volume manager to provide mirrored disk volumes [vxvm93a, vxvm93b], and these volumes are made resilient to node failure by providing remote mirrors, using a remote driver access mechanism [Armand91]¹. In the event of a node failure, the volume is restarted on the node containing the remote mirror.

In order to provide a uniform device name space for volumes shared by nodes (global volumes), we have implemented a global device numbering scheme which allows a volume to be known by a globally persistent device number. After recovery by the backup node, the device is still accessible using the same device number.

Each node contains its own root file system, and hence its own independent file name space. By convention, we mount shared file systems on the same mount point name on each node, to present a shared global file name space. Files outside this shared name space are private to the node.

File systems that are created on global volumes can be accessed by all nodes by mounting the block device on a directory in the standard manner. Remote requests to the file system are forwarded to the server node using CHORUS IPC protocols.

A remote file system is mounted using the same command line arguments as if it were on a local device, and the kernel performs the appropriate internal remote connection protocols. If we used a scheme like NFS which specifies the host and pathname on the host, we would have to update user visible data such as /etc/mnttab to reflect the new host after failover. With our scheme, the user visible state remains the same, namely the block device name, mount point name and mount options.

Failover of a file system from a failed node requires the volume to be started on the backup node, and the file system to be recovered. We use the VERITAS VxFS file system [vxfs92] to provide fast recovery using its intent log. The recovery required for failover is slightly different to the normal recovery that fsck performs, since it must preserve state held by active requests that fsck could throw away, such as unlinked files that are still open.

A number of components are used to provide these mechanisms:

- A distributed failover manager (FOM), which maintains the status of resources, whether they are active, failed, or in the process of being recovered. The FOM also maintains a list of clients who are using the resource.

1. Although we provide this remote mirroring by software, the architecture can accommodate the use of multi-ported disks.

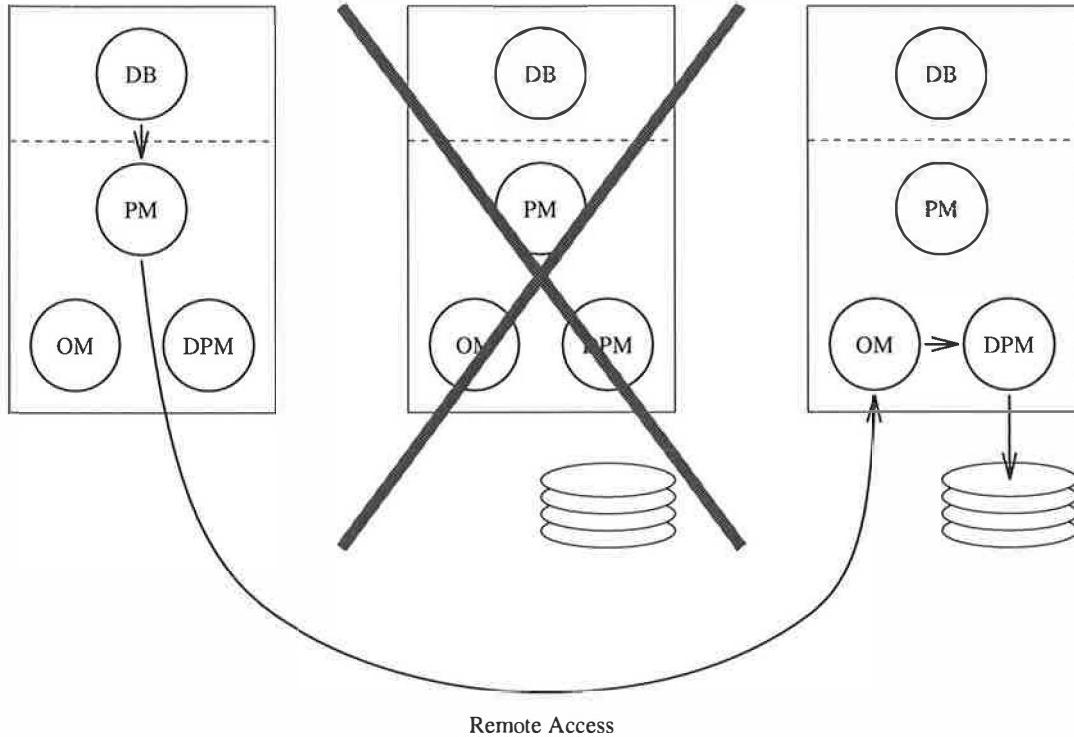


Figure 3. Access to a remote volume/filesystem after a failure

- A library linked with clients which handles the binding between clients and servers. The library is used to monitor IPC requests and detects failures using timeouts or by detecting link failures. After a failover, the library transparently rebinds the client to the new server.
- A state recovery protocol which allows servers to reconstruct state from information provided by clients.
- Modifications to server modules for server specific recovery actions.
- A mechanism for handling non-idempotent operations, to allow a backup server to determine if the operation was already completed before the failure.

4. Failure Detection and Handling

A set of resources and associated interfaces can be described as a *service*. At any given time a service is hosted by a single server, running on a particular node. An example of a service is a file system or raw disk device.

In general the servers in our system are the file Object Managers on each node, and clients are the Process Managers on each node. Object Managers may also be clients of a Process Manager, for example, to obtain process credentials, or to handle the /proc

file system operations [Killian84].

A service can be fault-tolerant or not; a fault-tolerant service is available (after some recovery action) even if the server currently hosting it fails. A service is the unit of failover, and if a particular server hosting multiple services fails, the individual services can be failed over to different backup servers. This can provide a better load balance between remaining nodes which pick up only a part of the activity of the failed node.

Services are identified by a unique identifier or name, in our case, a CHORUS unique identifier (UI) that is unambiguous across all nodes. This service UI is persistent during the lifetime of the system, and is persistent across a failover in the case of a fault-tolerant service.

The FOM is responsible for tracking the status of services, and maintains a list of clients using each service. The FOM is a global entity, distributed across all nodes. It is informed by servers when they first create a service, and by clients when they first access a service.

We provide a library, linked with each client, that monitors IPC requests. If these time out, or receive an error indicating the destination has failed, the FOM is notified.

The FOM determines whether the service has failed, and if it has, it informs all clients of the service, who then suspend all communication with the service. This is handled within the client-side library, and does not require any special action within the client.

It is important that a central agent determines the failure, as it is possible for conflicting error reports to be generated. For example, two nodes may be unable to communicate with each other, and each reports that the other has failed.

If the service is fault-tolerant, the FOM elects a suitable node to host the service, and instructs a server on that node to recover it. Once the recovery is complete, the clients are notified, and resume communication with the new server, including any requests that were blocked during the recovery.

If the service is not fault-tolerant, or the recovery failed, the clients are informed, and they mark the service as unavailable. This results in all requests returning with an error until the service is removed, for example, by unmounting a failed file system.

Servers are also prepared to handle the failure of their clients, detected by the fault detection library, and if necessary, clean up any state held on behalf of the client, for example, vnode references.

5. The Failover Manager

The Failover Manager (FOM) is responsible for detecting failures and initiating the failover procedure. It needs to be available at the time of a failure and hence needs to be resilient to node failure. The FOM is implemented as a set of daemons and actors. Some of these daemons run on an external host system.

The main components of the FOM are:

- *Error Recovery Manager* The ERM is the main interface for the kernel components and runs on the nodes. It is resilient to failure by the use of a number of hot backup ERM's running on other nodes. A master ERM is elected and serves as the consensus point for determining node failure and the coordinator of the failover process.
- *Disk Management Daemon* The DMD component knows which disk is connected to which node and where any second plex may be found. The DMD runs on the host.
- *Configuration Server* Conserv knows what the desired system should look like and what resources are critical to the users of the system

(e.g. the system should not continue if it loses both plexes of a specified volume). Conserv runs on the host.

Failure reports are sent by kernel components to the ERM, for example, a PM noticing an IPC timeout, the communications mechanism noticing a communications failure. When a failure has been established the ERM informs all clients of services on that node that the services have temporarily failed. Next, the ERM checks with Conserv if the system is still viable, (e.g. it hasn't lost any critical components). If the system is viable failover can proceed, otherwise the whole system needs to be shutdown and restarted. The ERM then informs the DMD of the node failure which launches the backup services on the site of the second plexes.

The host is involved with startup, shutdown and failure and is hence a critical component of the GOLDRUSH system; if it fails, no other node failure will not be recovered until the host has recovered. The host is a standard SVR4 machine that does not run any user services and can therefore be recovered by rebooting it².

6. Distributed State and Recovery

Each type of service imposes its own requirements for recovery. In general, the service must be recoverable to the state it was in before the failure occurred. This means that all volatile state maintained by the service must be identified, and somehow recreated in the backup server. Cached information does not need to be recovered, since this affects only the performance of the service.

Our strategy is to move as much of this volatile state as possible to the clients, and use some persistent storage for the rest. In the case of file systems, this includes such things as file and vnode reference counts, file seek offsets and directory blocks and so on. Apart from the file system specific data (incore inodes and so on), much of this state can be moved to the clients [Welch90].

During recovery, the file system can be recovered to a consistent state using fsck, and the reference counts and incore inodes can be recreated by recovering using client information [Baker94]. Once the volatile state has been reconstructed, the server can begin to receive new requests and retries of requests that were pending at the time of the failure.

2. The host is not a single point of failure. However permanent loss leaves the system unable to recover from node failure. A future goal is to fix this.

We use the VxFS file system, which uses transactions in an intent log to record all updates to file system structures. This provides very fast recovery, since completing the transactions recorded in the log will bring the file system to a consistent state. However, not all file system operations are atomic, and some minor changes were required to fsck to support failover. An example is unlinking an open file; the directory entry is removed, but the freeing of the inode is deferred until the last reference is released. During failover, this inode removal must be performed only once it has been determined that no remaining clients are referencing the file. The changes to fsck amounted to a new command line option, about 30 lines of code, and a new super block state that indicates that kernel level recovery is required to check reference counts of unlinked inodes.

Using a standard file system such as UFS [McKusick84] would create a number of problems. Firstly, without the atomicity of transaction-oriented disk update, it would be much harder to manage non-idempotent system calls. Secondly, the recovery would be much slower, since the fsck must scan the whole disk to repair the file system. In addition, the actions required to repair the file system can often result in the unpredictable loss of files, which complicates recovery.

Once the file system on disk has been made consistent, the kernel state must be recovered. Each client of the file system is contacted, and replies with reference counts and inode numbers of files it is using from the file system. These can be used to recreate the incore inodes.

A number of changes were made to the VFS interface [Kleiman86] to allow for file system specific actions during this recovery:

- When the file system is re-mounted, a new flag is passed indicating that this is for a failover. This allows the file system specific code to perform any special actions.
- A new VFS routine is used to recreate an incore inode and vnode with the correct reference counts and object capability.
- A new VFS routine is used to perform any file system specific activity once all the incore vnodes have been created. In the case of VxFS, this checks for deferred removals.

7. Non-idempotent Operations

Failed calls from a client to a server are retried when the failover service has resumed activity. Provision is made to prevent the system from executing the same non-idempotent requests twice.

Most existing systems use a server-side log which is duplicated on the backup server site. Instead we use a so-called “intent-message” mechanism with the support of the client side library.

When a non-idempotent request is received for the first time (ie. before a failure occurs), an upcall is performed to the client before committing the VxFS transaction on the inode. The “intent” message sent in this upcall identifies the request being processed, and contains a modification counter associated with the inode, as well as the expected return value of the request. The modification counter is then incremented, and the transaction is committed to disk, which also records the new counter value. The server then replies normally to the request.

If the transaction was completed before a failure, or if recovery rolls forward the transaction, the modification counter associated with the inode will have been incremented. If the client had not received the reply from the failed server, it will retransmit the request to the backup server; this retry request will contain the intent message sent by the failed server.

When the server receives this retry, it can determine whether the operation has already been completed, by comparing the modification counter of the inode with that in the message³; if the counters are the same, the operation had not completed, and is retried, otherwise the return value saved in the intent message is returned to the caller.

This simple mechanism has some interesting properties:

- Unlike log mechanisms, there is no log compression or cleanup issue, since the intent message information is kept with the client’s current request state. No extra allocation, deallocation, or management for the “log” memory is required.
- Since retry requests are modified with the contents of the intent message, there is no extra work for the server to determine whether this request is a retry of a non-idempotent request or not. There is no need to scan a server side log. All the information

3. The value of the counter at recovery time is used, since it may have evolved after recovery due to other intervening non-idempotent requests.

needed is within the message.

8. Comparison With Related Work

A number of other systems address the issues of high availability and fault tolerance in distributed environments. However, we believe our work differs from existing experiences in the following areas:

- Our system maintains the full UNIX semantics across distribution and recovery (stronger semantics than NFS based implementations).
- Client side caching is under control of the applications, not under control of the system, since applications are database servers.
- Replication is done at the logical volume level, not at the file level.
- We have based our implementation on commercially available software and have completed this mainly by introduction of a client side logging mechanism. The systems referenced below use mostly server side logging.
- We have introduced the notion of service and are able to deal with recovery of an individual service, using a generic failover manager which can be extended to services that are not necessarily file system based. None of the systems listed below appear to have worked in that direction.
- Our system supports (today) disk devices as well as filesystems.
- We do not rely on a new file system implementation (though VxFS was modified somewhat).

Sprite provides separates client and server state in a similar manner to our system [Welch90]. During recovery, each client must perform a re-open protocol on a per-file basis to provide the server with reference counts and so on, and to obtain a valid handle to the server object [Baker94]. However, in our system, the client handles are persistent, and the server only requires the reference counts and so on. This allows us to package the state for all files on the client into one RPC message, which reduces the amount of server congestion during recovery.

Spritely NFS [Mogul94] and Not Quite NFS [Mack94] are mostly based on NFS even though they extended NFS semantics. Anyhow, they do not make use of data mirroring, thus clients have to wait for the failed server to be up and running again before they can resume their activity.

HA_NFS [Bhide91] is closer to what we do for the failover mechanism. However it supports only NFS

requests, and requires some hardware so that the backup machine appears with the same IP address as the failed node. This also implies that all resources of the failed node will be backed by the same node.

DECEIT [Siegel90] is mostly based on NFS and thus has not had to deal with some of the issues we had to face, such as non-idempotent requests. DECEIT is also based on file replication rather than disk duplexing.

HARP [Liskov91] is based on file replication with a 2 phase-commit protocol from the primary machine to the secondary machine. A log is maintained in both sites and requires a Uninterruptible Power Supply. We have not found any detail on how clients detects failures and how they switch from primary to backup.

ECHO [Birrell89] and FICUS [Guy90] fall in the same category as the HARP file systems, being based on replication of files with secondary servers being synchronized with the primary one

CALYPSO[Devaraj94] is probably the closest system to ours, although there are a few differences. They use a three-phase recovery mechanism similar to our FOM, except that in GOLDRUSH, the state recovery is under the control of the backup server; the server can use the most appropriate mechanism for its recovery, for example, it can collect state from clients, or it can recover from a log filled by the crashed server. CALYPSO seems to be less flexible, requiring that server state is rebuilt from client caches. CALYPSO deals with the site crashes, and the activity of one site is taken over by another site; the notion of service as used in GOLDRUSH is a major improvement which allows the load of a crashed site to be spread among several surviving nodes.

CODA [Satya91] is mainly oriented towards disconnected operations and does not maintain UNIX semantics. Since most of the design is based on the existence of cache local to the client, the approaches are hardly comparable.

9. Performance Measurements

It has not been possible to completely isolate the effects of the fault tolerance mechanism as a large number of other changes have also been made over our previous system. We have been tuning the system with an emphasis on raw device access since this is the most performance critical for our applications. Our applications also mainly use read and write operations.

Call overhead for 512 byte raw device access			
Call	resilient access (ms)	standard access (ms)	overhead (%)
write	0.264	0.262	0.7
read	0.147	0.145	1.4
open	1.82	1.72	6.8

The overhead of using the fault detection library is minimal for the most frequent raw device accesses (around than 1% for read and write). However FOM interaction involves a larger overhead for open and mount (around 7%). This interaction involves communication with the ERM to declare the clients and services, and subscription to client and service failure. The ERM logs all operations to a file thus adding to this overhead.

The recovery time for a single raw volume is on the order of 2 seconds. The recovery time for VxFS is around 3 seconds for a 1GByte volume, and is proportional only to the size of the intent log. VxFS recovery is performed only after the raw volume has been sucessfully recovered.

The total recovery of a typical system (16 nodes, 10 user disks per node (duplicated), 10 volumes per duplicated disk group) after a node failure with the backup services being launched evenly across the remaining nodes is on the order of 20 seconds for raw volumes. VxFS recovery typically adds around another 10 seconds to this (typically only a few volumes have filesystems). This is single threaded on each node, but in parallel across the nodes.

10. Experiences and Further Work

Although our work utilised the inherently distributed nature of CHORUS/MiX we believe it can be adapted to other systems including monolithic kernels.

Although we had to port VxFS to the CHORUS/MiX environment, we found it had a number of useful properties - its transactional nature allows most operations to be treated atomically, simplifying recovery, and the intent log provides very fast recovery time, on the order of a few seconds.

The current system has been implemented and is currently undergoing system test. We are tuning the system with the empahsis on raw device access performance since this is the most critical for our applications.

We have yet to fully measure the performance of the system and are in the process of developing our measurement techniques to isolate the overhead of non-

idempotent calls.

We found the performance of frequently used operations to be reasonable. It will be possible to further improve performance by moving the ERM into kernel space as a CHORUS supervisor actor, thus alleviating a large number of context switches especially on mounting a filesystem or the first open of a raw volume.

11. Acknowledgements

We would like to thank the efforts of the people who contributed to the design and implementation of the mechanisms described in this paper: Brian Anthony, Richard Harry, Martin Hogg, Simon McKenna, David Messham, Steve Noble and Iain Robertson at ICL, and Jean-Marc Fénart, Ruby Krishnaswamy, Pierre Lebée and Gilles Maigné at Chorus Systèmes.

12. Biographies

Sunil Kittur received his BSc in Computer Science in 1988 from University College London. He spent the next 4 years at the Santa Cruz Operation, working on the SCO XENIX, UNIX and MPX kernels. In 1992 he joined ICL High Performance Systems as a senior engineer on the GOLDRUSH project. He has recently joined Online Media as principal software engineer working on the OS for a distributed interactive multi-media system. His email address is skittur@omi.co.uk.

François Armand received his Engineer diploma in 1977 from ENSEEIHT in Tolosa, France. He spent a few years in a software house and joined the Sol research project at INRIA, working on Unix V7, in 1980. He then moved to the Chorus research project in 1985, and since 1987 has been at Chorus Systems working on aspects of the Unix subsystem for the CHORUS microkernel. He has been mostly involved in the design of the distributed Unix and more recently in failure resilience issues. His email address is francois@chorus.fr.

Douglas Steel completed his BSc in Computing Science in 1988 from Glasgow University. He spent 4 years as a research assistant at Queen Mary & Westfield College (University of London) investigating operating system support for distributed object oriented programming, and completed a part-time MSc. He joined Unix Systems Laboratories Europe and worked on the Esprit Ouverture project. He joined ICL High Performance Systems in late 1993 as a senior engineer on the GOLDRUSH project. His email address is doug@wg.icl.co.uk.

Jim Lipkis has been involved in operating system design for parallel, real time, and fault tolerant systems. At New York University he worked on OS and language software for scalable shared-memory multiprocessors such as the NYU Ultracomputer. Since 1989 he has been a senior engineer and architect at Chorus Systems, working on microkernel design and on application of the microkernel to systems ranging from embedded real time to supercomputing to highly available parallel database servers. His email address is lipkis@chorus.fr.

REFERENCES

- [Armand91] F. Armand, "Give a Process to your Drivers!", Proc. of the EurOpen Autumn 1991 Conference.
- [Baker94] Mary Baker, "Fast Crash Recovery in Distributed File Systems", PhD Thesis, Univ. California at Berkeley, 1994.
- [Batlivala92] Nariman Batlivala et al., "Experience with SVR4 Over Chorus", Proc. of the USENIX Workshop on Micro-Kernels and Other Kernel Architectures, April 1992, pp.223-242.
- [Bhide91] Anupam Bhide et al., "A Highly Available Network File Server", Proc. of the Winter 1991 USENIX Conference, pp.199-218.
- [Birrell89] A. Birrell et al., "Availability and Consistency Tradeoffs in the Echo Distributed File System", Proc. of Second Workshop on Workstation Operating Systems, pp.49-54.
- [Batlivala92] N. Batlivala et al., "Experience with SVR4 over Chorus" Proc. of 1st Workshop on Microkernels and Other Architectures, Usenix, Seattle 1992.
- [Devara94] M. Devarakonda , B. Kish, A. Mohindra "Non-Disruptive Server Recovery in Calypso File System" IBM Researh report RC 19794(87665) 10/19/94
- [Guy90] Richard G. Guy et al., "Implementation of the Ficus Replicated File System", Proc. of Summer 1990 USENIX Conference, pp.63-72.
- [Killian84] T.J. Killian, "Processes as Files", Proc. of the Summer 1984 USENIX Conference.
- [Kleiman86] S.R. Kleiman, "Vnodes: An Architecture for Multiple File System Types in Sun UNIX", Proc. of the Summer 1986 USENIX Conference, pp.238-247.
- [Liskov91] Barbara Liskov et al., "Replication in the Harp File System", Proc. of the 13th ACM Symposium on Operating Systems Principles, pp.226-238.
- [McKusick84] M.K. McKusick et al., "A Fast File System for UNIX", ACM Transactions on Computer Systems Vol.2 No.3, August 1984, pp.181-197.
- [Mack94] Rick Macklem, "Not Quite NFS, Soft Cache Consistency for NFS", Proc. of the Winter 1994 USENIX Conference, pp.261-278.
- [Mogul92] Jeffrey C Mogul, "A Recovery Protocol for Spritely NFS", Proc. of the USENIX File Systems Workshop, May 1992, pp.93-110.
- [Mogul94] Jeffrey C. Mogul, "Recovery in Spritely NFS", Computing Systems, Spring 1994, pp.201-262.
- [Rozier] M. Rozier et al., "Chorus Distributed Operating Systems" Computing Systems 1(4), December 1988.
- [Sandberg85] R. Sandberg et al., "The Design and Implementation of the Sun Network File System", Proc. of the Summer 1985 USENIX Conference, June 1985, pp.119-130.
- [Satya91] M. Satyanarayanan et al., "Disconnected Operation in the Coda File System", Operating Systems Review, vol. 5, pp.213-225.
- [Siegel90] Alex Siegel, "Deceit: A Flexible Distributed File System", Proc. of Summer 1990 USENIX Conference, pp.51-62.
- [vxfs92] Veritas Sofware Corporation, "VERITAS File System (VxFS) System Administrator's Guide Release 1.2.1", 1992.

- [vxvm93a] Veritas Software Corporation,
"VERITAS Volume Manager
(VxVM) Basic User's Guide
Release 1.2", 1993
- [vxvm93b] Veritas Software Corporation,
"VERITAS Volume Manager
(VxVM) System Administrator's
Guide Release 1.2", 1993
- [Welch90] Brent B Welch, "Naming, State
Management, and User-Level
Extensions to the Sprite Dis-
tributed File System", PhD Thesis,
Univ. California at Berkeley, 1990.

FLIPC: A Low Latency Messaging System for Distributed Real Time Environments

David L. Black,¹ Randall D. Smith, Steven J. Sears,² and Randall W. Dean

Open Software Foundation Research Institute, Cambridge, MA
dlb@osf.org, randys@osf.org, sjs@novell.com, rwd@osf.org

Abstract

FLIPC is a new messaging system intended to support distributed real time applications on high performance communication hardware. Application messaging systems designed for high performance computing environments are not well suited to other environments because they lack support for the complex application structures involving multiple processes, threads, and classes of message traffic found in environments such as distributed real time. These messaging systems also have not been optimized for medium size messages found in important classes of real time applications. FLIPC includes additional features to support applications outside the high performance computing domain. For medium size messages, our system significantly outperforms other messaging systems on the Intel Paragon. An explicit design focus on programmable communication hardware and the resulting use of wait-free synchronization was a key factor in achieving this level of performance. The implementation of FLIPC was accelerated by our use of PC clusters connected by ethernet or by a SCSI bus as development platforms to reduce the need for Paragon time.

1 Introduction

Messaging systems for high performance computing are designed to deliver as much of the hardware performance of the interconnect to applications as possible. Examples include the NX system on the Intel Paragon [11], CMM on the Thinking Machines CM-5 [17], and Active Messages on various platforms [2][19]. These systems have been optimized for numerical supercomputing to the detriment of their use in

other environments. Among the factors that contribute to this limitation are the assumption of a single threaded application structure, system optimizations for small and large message sizes to the detriment of intermediate sizes, and the assumption of a single application environment in which there are no competing sources of message traffic. Our work concerns distributed real time environments in which these assumptions do not hold.

Continuing advances in networking technology, such as ATM [18], and Myrinet [1], have dramatically increased the communication performance available to environments other than numerical supercomputing. This paper describes our experimental implementation of a high performance messaging system for one such environment, namely event driven distributed real time. This environment is characterized by the need to support multiple application threads of varying importance or priority, and messaging streams of varying importance concurrently on each node. This includes not only processing, but also resource allocation. For example, the system must not only process a message announcing detection of an incoming message in preference to a message indicating that it is time for preventative maintenance, but must also ensure that the latter message does not consume resources required to handle the former.

The event driven nature of applications in this environment results in messages that fall between the small and large sizes found in numerical supercomputing. The events cannot be described by very small messages, and aggregation of events into larger messages is limited by the impact of the aggregation delay on system response. Examples of such systems include distributed systems for process control, factory floor automation, and military command and control [5] (e.g., AEGIS, AWACS).

1. This research was supported in part by the Advanced Research Projects Agency (ARPA) and the Air Force Material Command (AFMC).

2. Steven J. Sears is currently with Novell, Inc.

The FLIPC messaging system is designed for event driven real time environments. Among its more important characteristics are:

- performance optimization for medium sized messages (50-500 bytes),
- support for multithreaded applications,
- support for threads and message streams of varying importance,
- support for explicit control of resource allocation.

2 Communication Interface Architecture

FLIPC is designed to leverage the powerful programmable controllers that are increasingly found in the interfaces to high speed interconnects and networks. Examples of these controllers include the dedicated message processor on each node of a Paragon system [6], embedded microprocessors in communication controllers such as the i960 [4], and custom designed communication controllers such as Myrinet [1], SCSI [10], and FLASH [8]. The increasing use of such controllers is motivated by the need to match increases in processor speed with increases in communication performance, and the resulting desire to off-load communication functionality from the main processor(s). Similar trends have been observed in other domains, such as mainframe input/output and graphics controllers[14]. Experience in these domains suggests that the power and functionality of the programmable controllers in network interfaces will continue to increase.

The power of these interface controllers cannot be directly utilized by applications because the controllers and their execution environment are functionally specialized to communication tasks. The resulting obstacles to execution of application code include:

- **Instruction Set Differences:** Of the controllers cited above, only the dedicated Paragon processor employs the same instruction set as the application processor(s).
- **Protection Concerns:** Controllers have access to critical system resources, often including all of physical memory. Communication is also critical to system operation, as a controller crash may imply a node crash. A related problem is that an application must be prevented from monopolizing the communication resource to the exclusion of other applications. As a result, untrusted code cannot be executed on such interface controller.
- **Memory Access Restrictions:** Neither the SCSI nor Myrinet controllers are capable of performing

read-modify-write atomic operations such as test-and-set on main memory.

- **Execution Restrictions:** A common structure for controller software is a non-preemptible event loop. This puts restrictions on the time that may be consumed by added code; excessive consumption may have undesirable side effects on unrelated communications.

FLIPC's architecture takes advantage of interface controllers by programming them as operating system components rather than as part of the applications. This enables the use of system programming tools that target the embedded controller, and simplifies protection concerns. Restrictions on memory access and execution are dealt with in the design of the FLIPC component that executes on the controller. The alternate approach of using software fault isolation [20] is of limited utility because correctness of the code executing on the controller depends not only on the memory it accesses, but also on completing execution in a timely fashion. Restrictions such as forbidding backwards branches are necessary, greatly reducing the programming flexibility.

3 Message Sizes

The design of FLIPC is based on the observation that there are three distinct message size classes with corresponding distinct implementation techniques for high performance. The classes and corresponding implementation techniques are:

- **Small.** Small messages can both be stored in registers and be copied without seriously impacting performance in either case. Current technology puts an upper bound on their size at about 32 bytes on a 32 bit processor with 32 registers. Implementation techniques for this class include carrying a message in registers and allowing the message to be copied to avoid coupling application and message transport interfaces. Hardware DMA functionality may not be useful for this class of communication because the messages are too short to amortize DMA setup costs.
- **Medium.** Medium messages are too large to fit in registers or to be copied without impacting performance, but too small to amortize the setup cost of a throughput-oriented transfer protocol. The result is a size range from around 40 bytes to a kilobyte or small number of kilobytes. Direct access to application memory and the use of DMA support, if available, are important to achieving performance.

- **Large.** The lower bound of this class of messages is the size at which the primary performance concern shifts from message latency or message throughput to data throughput within a single message. Optimizing for data throughput requires a protocol that maximizes use of the available bandwidth without causing receiver overrun. This necessitates transfers directly to and from application memory, and may require flow control. The resulting protocols often have a setup phase to validate the memory addresses on the remote system and initialize any required flow control. Messages using such a protocol must be large enough to amortize this setup overhead. This minimum size may be a kilobyte or more.

With the exception of Express Messages [9], most other messaging systems have recognized only the small and large message classes. FLIPC is designed to optimize performance of the medium class of messages by using a shared memory interface to avoid copying, an optimistic basic transfer protocol designed to move this class of messages, and a flow control architecture that does not involve the basic transport protocol.

4 Architecture and Design

FLIPC contains three major components, as shown in Figure 1:

- The *messaging engine*. This is the body of hardware and software that moves messages between nodes.
- A fixed size non-pageable *communication buffer* that is shared between the messaging engine and all applications that use FLIPC.

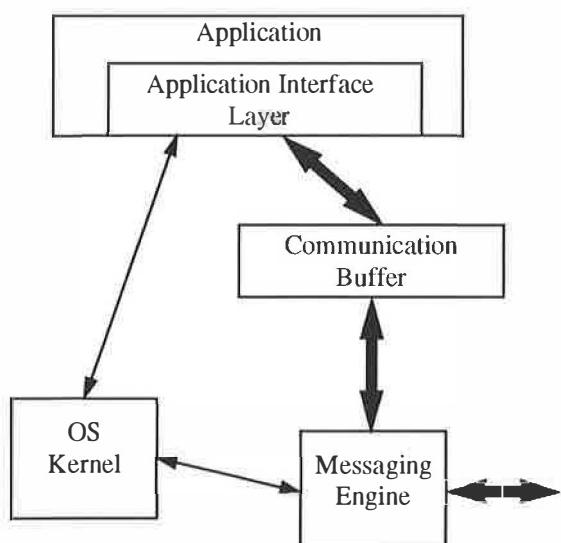


Figure 1: FLIPC Architecture

- An *application interface layer* that provides formal interfaces to applications and hides the data structures in the communication buffer. This consists of both a library and header file(s).

The *operating system kernel* is involved only in synchronization actions that cannot be directly accomplished via state in the communication buffer. The heavy arrows in Figure 1 indicate message flow, and the light arrows indicate synchronization operations that require interaction with the operating system kernel.

The messaging engine is an independently executing component of the system. It is intended to execute on the programmable controller in the communication interface when one is present, but can also be implemented as part of the operating system kernel for debugging purposes or on systems lacking the required hardware. Synchronization between the messaging engine and the application consists entirely of wait-free synchronization, making it impossible for an errant application to stall the communication controller.

The communication buffer is the focal point of FLIPC. It is located in shared memory accessible to both the application(s) and the messaging engine, and it contains all of the memory resources used for messaging. This design enables direct interactions between the application and the messaging engine without requiring code from either component to cross the protection boundary into the other. An additional result is that the OS Kernel is removed from the messaging path, making that path significantly shorter. Protection of the messaging engine from the application can be enforced via appropriate checks in the messaging engine, but can be removed to increase performance of a trusted application.

FLIPC implements multiple endpoints per communication buffer to support application requirements. Using different endpoints for different threads avoids contention for communication resources in multi-threaded applications. Multiple cooperating applications can run on a single node by dividing or otherwise sharing the endpoints in a single communication buffer. The implementation of resource control at the endpoint level makes it easy to separate resources for different classes of traffic by using different endpoints.

An endpoint group logically combines multiple endpoints into a single abstraction. FLIPC supports a receive operation that retrieves a message from an endpoint if there is an available message on any endpoint in the group. This operation is implemented entirely in the library because the resource control

model's association of buffers with endpoints makes it infeasible to merge the endpoint buffer queues. FLIPC also supports a blocking receive operation on an endpoint group. This design approach was chosen over the interrupting upcall methodology of active messages because application level interrupts are not appropriate for a multitasking real time environment. Interrupts disrupt execution in a way that cannot be controlled by the scheduler, reducing the real time predictability of the system. In contrast, FLIPC provides a real time semaphore option that causes the thread awakened by a message arrival to be presented to the scheduler in the OS kernel, allowing it to determine when it is appropriate to execute that thread.

Fixed size messages simplify the messaging engine logic. The actual size restriction is platform dependent. For the Intel Paragon, the characteristics of the DMA support in the interconnect interface require a message size that is at least 64 bytes and a multiple of 32 bytes. FLIPC uses 8 bytes of each message for internal addressing and synchronization purposes, so 56 bytes is the minimum application message size. Transfer of messages larger than the fixed size selected at boot time is not supported. FLIPC shields applications from buffer alignment restrictions by internalizing all message buffers. An application must call FLIPC to allocate a message buffer, allowing the implementation to ensure that all such buffers are correctly aligned.

Addressing facilities were designed to minimize implementation complexity. FLIPC message destinations (receive endpoint addresses) are opaque and determined by the system. This requires receivers to obtain endpoint addresses of endpoints they have allocated from FLIPC and pass those addresses to senders. FLIPC does not contain a nameservice of its own, but assumes that one is available for this purpose.

5 Message Transfer

FLIPC's basic messaging operation performs an asynchronous one way delivery of a fixed size message from a source endpoint to a destination endpoint. The send and receive interactions with the messaging engine are symmetric in that both involve queueing a message buffer for the messaging engine and subsequently determining whether the send or receive operation has completed. Both polling and blocking versions of completion detection are supported.

Figure 2 shows the complete sequence of operations involved in a message transfer. Step 1 involves the receiver providing a message buffer to receive the message. The sender sends the message at step 2 by queueing the message buffer on the endpoint for the

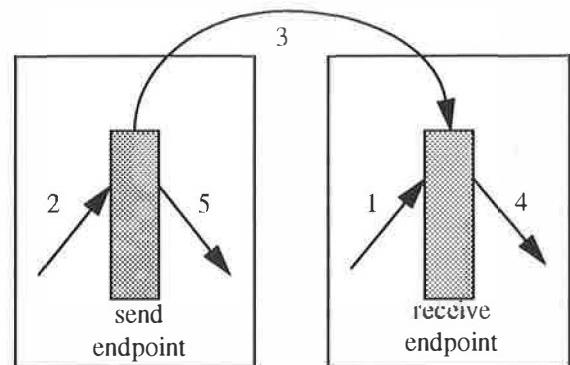


Figure 2: FLIPC Message Transfer

messaging engine. The messaging engine transfers the message in step 3, queueing it on the receive endpoint by placing it in the buffer supplied at step 1. The receiver receives the message by removing it from the receive endpoint at step 4, and the sender recovers its buffer for reuse by removing it from the send endpoint at step 5. Steps 2 through 4 are the actual message delivery path, with steps 1 and 5 being associated resource control operations.

FLIPC uses a message delivery model that combines a reliable ordered transport with higher level responsibility for resource management and any associated flow control. The messaging engine provides a reliable transport that preserves order for messages sent from the same source endpoint to the same destination endpoint. Vesting responsibility for resource management and flow control in the application and the application library allows the transport to ensure that every node is always prepared to receive from the interconnect. If a receive occurs without an available buffer on the destination endpoint, the received message is discarded. The resulting guarantee that all messages in transit can always be accepted by their receiving node avoids deadlocks on reliable interconnects. FLIPC maintains a counter in each endpoint to track discarded message events, and makes this available to applications.

Flow control to avoid discarded messages can be provided either by applications or by libraries designed to fit between applications and FLIPC. This structure greatly simplifies the buffer management logic in FLIPC and allows flow control policies to be customized to application needs. In some cases, static properties of the application structure may remove the need for runtime flow control. One example is that an RPC interaction structure with a fixed set of clients can statically determine the number of buffers needed based on the maximum number of clients. Another example

is that an application made up of strictly periodic components can often determine its worst case buffering needs in advance based on the maximum number of messages sent per time period.

6 Wait-Free Synchronization

Properties of the programmable controllers that FLIPC exploits require the use of wait-free techniques for synchronization between the applications and the messaging engine. This is necessitated by the fact that the controller on which the messaging engine executes may be the only means of communication with the rest of the system. Hence, a controller hang may render the node useless. Wait-free synchronization ensures that an ill-behaved application cannot stall the controller and cause damage in this fashion.

This synchronization problem is further complicated by the inability to assume that the programmable controller can execute atomic memory operations other than loads and stores. The resulting memory model is the same as that assumed by mutual exclusion algorithms such as Peterson's algorithm [15]. We simplified these synchronization problems by making the combination of the application and the application library component of FLIPC responsible for all synchronization among application threads. This reduces the wait-free synchronization problems to two-way problems between the messaging engine and an application thread. We solved these by separating or duplicating data structure components so that the application and messaging engine never attempt to concurrently write the same memory location. The synchronization among application threads is implemented via conventional multithreaded locking techniques based on a test and set lock, because these threads cannot execute on the programmable controller.

This type of wait-free synchronization requires a different approach to designing data structures because the only available atomic operations are reads and writes. For example, consider the counter used to record the number of discarded messages for each endpoint. FLIPC allows the application to read and reset this counter as a single logical operation, and guarantees that the reset will not cause dropped message events to be lost. A single memory location is not sufficient to implement a wait-free version of this counter because message drops occurring between the read and the write that resets the counter to zero will be lost. Instead FLIPC uses two memory locations to implement the counter. The first location is incremented each time a message is discarded, and the second location

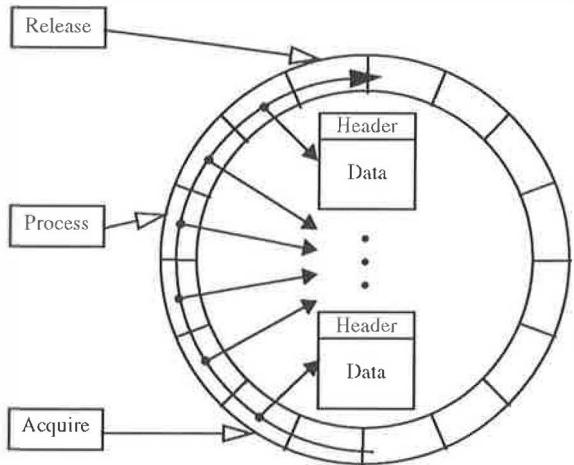


Figure 3: Endpoint Buffer Queue

holds the value of this count as of the last 'read and reset' operation. The actual count of discarded messages is then the difference between these two values, and the reset is implemented by copying the value from the first location to the second location. This cleanly separates the data structure into one location written only by the messaging engine, the counter, and one written only by the application, the copied value.

The most important data structure for synchronization between the application and the messaging engine is the buffer queue in each endpoint. As shown in Figure 3, this structure consists of a circular queue of buffer pointers that tracks not only the queue head and tail, but also how far the messaging engine has proceeded through the queue. The solid circular arrow in the figure indicates the direction in which the release (head), process (middle), and acquire (tail) pointers move as buffers move through the endpoint. The application releases buffers to the endpoint by inserting pointers to them at the front of this queue. The messaging engine uses the process pointer to follow behind the front of the queue, sending from or receiving into these buffers depending on the endpoint type. Buffers processed by the messaging engine become free to be acquired by the application. This buffer acquisition occurs at the tail of the queue. The queue is empty when all three pointers point to the same location; the two half empty conditions of no buffers to process and no buffers to acquire occur when the corresponding two pointers point to the same cell. Each buffer also contains a state field that is changed when processing has been completed, allowing an application to determine when processing of a specific buffer is complete.

7 Implementation

FLIPC has been implemented on three different hardware platforms. The application interface library and communication buffer data structures are the same in all three cases, demonstrating the generality and portability of the system. On the other hand, the messaging engine is necessarily specific to the communication hardware and requires reimplementation for each platform. Our development platforms have been PC clusters interconnected by ethernet or a SCSI bus [3]. We have also implemented FLIPC on the Intel Paragon using the Paragon mesh interconnect. Our initial development employed transports for all three platforms developed by another project as OS kernel components to a common interface, the Kernel to Kernel Transport (KKT) interface [13]. This interface is not a good match to the one way messages used by FLIPC because KKT uses an RPC to deliver each message. On the other hand, this was very effective for development purposes, because it allowed us to focus on the platform independent components of FLIPC. The resulting development strategy allowed us to build and debug a fully functional system without using scarce Paragon time. When Paragon time became available, we were able to move the KKT implementation of FLIPC from the SCSI and ethernet clusters to the Paragon in less than a week including test time.

We then developed a native FLIPC messaging engine optimized and customized for the Paragon hardware, specifically Paragons that use MP3 nodes. These nodes contain three 50MHz i860 processors, one of which is reserved as a message coprocessor for accelerating messaging operations. Cache coherency is implemented among all three processors. This messaging engine executes directly on the Paragon's message coprocessor and accesses the communication buffer directly instead of using the KKT transport interface. The inter-node protocol for message transfer is an optimistic protocol that aggressively sends messages without acknowledgment or feedback. As noted previously, the receiving node discards messages if receive buffers are not available. This protocol coexists with other protocols in the Paragon's protocol framework on the message coprocessor, allowing multiple protocols to be used simultaneously³. For instance, our implementation of FLIPC on the OSF/1 AD operating system [22] requires both the FLIPC and OSF/1 AD

protocols to operate simultaneously. Section 8 reports performance results for this implementation.

Tuning this implementation uncovered two performance problems caused by the cache coherency implementation on the multiprocessor Paragon nodes. The first problem was that the caches do not implement cache residency for multiprocessor locks. Instead, the test and set operation that acquires a lock must lock the bus and perform the read and write operations directly on memory, with a severe impact on performance. We avoided this problem by implementing versions of the message send and receive operations that do not use multiprocessor locks for mutual exclusion among application threads. These operations are intended for use by applications whose structure ensures that at most one thread will access each endpoint, or for which mutual exclusion can be provided at a higher level. All of our performance results use these interface versions. The second problem was that false sharing of variables written by both the messaging engine and the application library in the same 32 byte cache line on the Paragon caused excessive numbers of cache invalidations. We solved this problem by extending our design approach to wait-free synchronization to ensure that concurrent writes from the application and messaging engine can never occur in the same cache line. This reorganization of communication buffer data structures eliminated the false sharing. The combination of these two optimizations improved latency by 15µs or almost a factor of two. We had expected to encounter cache effects in tuning FLIPC, but were surprised by the magnitude of their impact.

8 Performance

Figure 4 shows the message latency for the optimized FLIPC implementation on the Paragon. The measured message latencies range from about 15.5µs to 17µs.

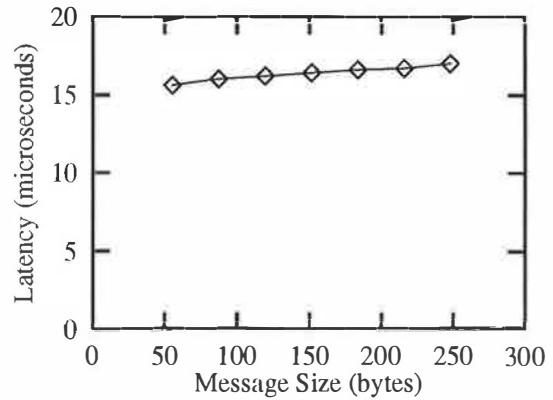


Figure 4: FLIPC Message Latency

3. The overall structure that makes it possible to integrate the FLIPC protocol into the Paragon's protocol framework is based on work originally done by Lok Liu as part of the Paragon Active Messages project [2].

The corresponding standard deviations range from $0.5\mu\text{s}$ to $0.65\mu\text{s}$ and are approximately the size of the symbols used in the figure. For message sizes of 96 bytes and above, the latencies obey the following equation:

$$\text{Latency} = 15.45\mu\text{s} + 6.25\text{ns}/\text{byte}$$

Shorter messages can be sent slightly faster due to changes in hardware behavior. The slope of $6.25 \text{ ns}/\text{byte}$ indicates that increasing the FLIPC message size increases the use of interconnect bandwidth at over 150 MB/s. This is on an interconnect whose hardware peak bandwidth is 200 MB/s, and for which the best throughput achieved by any software is 160 MB/s [12].

These measurements were obtained via a test program that measures the time consumed by multiple two-way message exchanges between a pair of nodes. The time for a single message is then obtained by dividing this overall time by twice the number of two-way exchanges. These results are from a configuration that does not contain all of the validity checks that protect the messaging engine against corruption of the communication buffer by an errant or malicious application. Configuring these checks adds an additional $2\mu\text{s}$ to the above times.

Running the test program for a small number of exchanges yields results that are about $3\mu\text{s}$ faster than the above steady state results from test runs that include hundreds of message exchanges. We believe that the faster results for small numbers of exchanges are explained by cache start-up transients. The 16 kilobyte caches used by the i860 processors on the Paragon are relatively small, and the Paragon does not implement any secondary caches. The result is that the caches are sensitive to disturbances caused by executing code outside the inner test loop, so that saving the results of the previous test cycle is sufficient to cause replacement of a significant portion of the cache. The result of these replacements is that cache lines that are shared in the steady state of the test are not shared at start-up, resulting in fewer cache invalidations. This performance anomaly reinforces the importance of cache effects as an impact on performance of this class of messaging systems.

9 Related Work

FLIPC is most closely related to two systems, Paragon Active Messages (PAM) on the Paragon and Express Messages on the iPSC/2 hypercube.

PAM [2] consists of two communication subsystems, an active messages facility that transfers 20 byte messages, and a bulk transport facility based on direct read and write operations to remote memory. The active messages facility is layered on top of an optimistic transport protocol for fixed size messages via polling for incoming messages. PAM's optimizations for small messages and the simpler functionality by comparison to FLIPC yield a message latency of less than $10\mu\text{s}$, about a third faster than FLIPC would be on a 20 byte message. An important difference caused by optimizing for small messages is that application preallocation and management of message buffers is not needed by PAM because a 20 byte message can be copied to or from an internal data structure at almost zero cost, less than $0.2\mu\text{s}$. Functionality differences include the support for multiple endpoints, data streams, and the related resource control that is found in FLIPC, but not PAM. PAM's bulk transport mechanism is complementary to FLIPC, as FLIPC does not contain a bulk transport mechanism.

There are strong similarities between the PAM and FLIPC implementations in that both use a wired communication buffer shared between the application and messaging engine, and an optimistic transport that discards messages when receive resources are not available. Both systems recognize that a dedicated buffer avoids deadlock problems that may arise in the use of an optimistic transport on a reliable interconnect, although the actual buffer designs are quite different. Both systems also move flow control support from the basic internode protocol to higher layers to support customizing flow control to application requirements. The window based flow control protocol used by PAM's active message facility is an example of this customization. PAM is targeted to smaller messages than FLIPC; PAM messages are 28 bytes, of which 8 bytes are used by PAM, leaving 20 for the application. PAM implements active messages on top of its transport by using 4 bytes of the message to hold the address of the remote message handler; the same implementation could be used on FLIPC.

Express Messages [9] was a messaging facility for the iPSC/2 hypercube. This system contained several ideas that were utilized and enhanced in the design of FLIPC. Express Messages recognized the distinction among small, medium, and large messages, and also used an aggressive optimistic transfer protocol for medium messages. Fixed size message buffers were used for medium messages, but via page mapping techniques instead of a shared memory buffer. A shared control bit was used switch between polling and interrupt-based message delivery mechanisms, but system

calls were used for buffer management in contrast to the shared data structure implementation in FLIPC. The absence of kernel threading support in the NX/2 operating system required Express Messages to implement threading at user level, including a handoff mechanism that executed an interrupted critical section if the interrupt handler needed to enter the section. FLIPC is based on kernel thread support, allowing it to deliver messages to threads rather than upcall handlers. This avoids interrupting critical sections. Instead, the message that would have caused an interruption is handled by a thread, allowing conventional multithreaded synchronization techniques to be used to resolve critical section conflicts.

Two other common high performance messaging systems for the Paragon are NX [11] and SUNMOS [21]. Both optimize performance of large messages for high performance numerical computation, and SUNMOS also optimizes zero length messages. NX is part of the basic Paragon operating system, whereas SUNMOS is a single application operating system that is run on a subset of the Paragon nodes, utilizing the Paragon operating system running on the remainder of the machine for the services it does not provide. NX achieves a bandwidth⁴ of over 140 MB/sec and SUNMOS approaches 160 MB/s for sufficiently large messages. Aside from the underlying hardware architecture, FLIPC has little in common with these systems because they are optimized for large messages rather than medium messages. FLIPC is complementary to such systems, as a bulk transfer mechanism needs to be added to FLIPC to obtain a complete system. SUNMOS has the further complication that its basic messaging protocol assumes a non-multiprogrammed system by sending multi-megabyte messages as a single packet. This occupies the path through the interconnect for the duration of the message and is a potential responsiveness problem in a real time environment.

The optimized FLIPC implementation for the Paragon significantly exceeds the performance of these other messaging systems on medium sized messages. The FLIPC latency for a 120 byte message transferred between applications on two Paragon nodes is 16.2µs. The comparable latencies for the three other messaging systems on the Paragon are:

- NX (Paragon O/S R1.3.2), 46µs [11].
- Paragon Active Messages, 26µs [2].

4. The NX performance results reported in [2] are from an older release of NX. This result is from a more recent version (1.3.2).

- SUNMOS, 28µs [12][21].

This demonstrates the performance impact of not optimizing for the medium class of messages. These three messaging systems have been optimized for bandwidth on large messages. In addition, PAM has been optimized for latency on very small messages.

10 Future Work

There are a number of opportunities for future work to improve on FLIPC. The explicit resource management model is effective, but is also awkward to program. Our experience is that a FLIPC application can expect to employ about half of its calls to FLIPC to send or receive messages, and the other half for message buffer management. An improved buffer management design that frees the programmer from most of these details is clearly called for. Support for multiple communication buffers per node and protection mechanisms that restrict where messages can be sent should be added to support multiple applications that do not trust each other. FLIPC was designed solely to address the transport of medium sized messages and needs to be integrated into a system that provides excellent performance for messages of all sizes. As part of this work, we are considering extensions that allow applications to indirectly access memory on other nodes [16]; some related ideas can be found in the SUNMOS [21], PAM [2], and Illinois Fast Messages [7] systems. Finally, we intend to pursue further integration of FLIPC into a real time environment by adding real time prioritization and capacity/bandwidth control functionality to the basic inter-node transport.

11 Conclusion

FLIPC incorporates an innovative combination of design and implementation techniques that achieve excellent performance for medium messages on high speed communication hardware in a real time environment. The most important architectural approach is to recognize and explicitly target programmable communication controllers. The FLIPC design manifests this targeting in several ways:

- Logical separation of messaging engine and OS kernel allows the messaging engine to be implemented on the communication controller.
- Asynchronous messaging interfaces and wait-free synchronization decouple the communication controller from applications.
- The shared communication buffer allows the OS kernel to be bypassed.

FLIPC is designed for medium messages, an important class of messages for distributed real time applications. The increased performance it obtains by comparison to other messaging systems indicates that this size class of messages has been overlooked by these other systems.

Our experiences in implementing FLIPC provides some useful guidance for designers of other systems. The importance of cache management to the performance of distributed messaging systems is indicated by the cache problems we encountered during performance tuning, and the factor of two in performance obtained by tuning for cache effects. The limitations of some programmable communication controllers required us to address wait-free synchronization problems in a memory model without atomic read-modify-write operations. Finally, employing PC clusters proved to be an effective way to develop prototypes, including validating the portability of FLIPC across two different communication mediums. This greatly reduced our need for Paragon time to develop the final version of FLIPC, because the platform independent components had already been implemented and debugged.

Acknowledgments

For their assistance in making this work possible, we are grateful to the Intel Scalable Systems Division, especially Paul Pierce and Roy Larsen. We would also like to thank the Active Messages group of the University of California at Berkeley, and especially Lok Liu for making the source code of Paragon Active Messages available for us to study. Paul Davis of Honeywell kindly provided measurements on the current version of NX. Dejan Milojicic and the anonymous Usenix reviewers provided comments that greatly improved this paper.

References

- [1] Nanette J. Boden, Danny Cohen, Robert E. Felderman, Alan E. Kulawik, Charles L. Seitz, Jakov N. Seizovic, and Wen-King Su, 'Myrinet -- A Gigabit-per-second Local-Area Network', *IEEE Micro*, February 1995.
- [2] Eric A. Brewer, Frederic T. Chong, Lok T. Liu, Shamik D. Sharma, and John Kubiatowicz, 'Remote Queues: Exposing Messages for Optimization and Atomicity', *Proceedings of the 7th Annual ACM Symposium on Parallel Algorithms and Architectures*, July 1995, pp. 42-53.
- [3] Randall W. Dean, Michelle M. Dominijanni, Steven J. Sears, and Alan Langerman, 'SCSI for Host to Host Communication', OSF Research Institute Technical Report, In preparation.
- [4] Peter Druschel, Larry L. Peterson, and Bruce S. Davie, 'Experiences with a High-Speed Network Adapter: A Software Perspective', *1994 SIGCOMM Conference Proceedings*, October 1994, pp. 2-13.
- [5] 'High Performance Distributed Computing (Hiper-D) Integrated Demonstration One Report', Naval Surface Warfare Center (NSWC), Dahlgren, Virginia, September 1994.
- [6] Intel Corporation, *Intel Paragon XP/S Supercomputer Spec Sheet*, 1992.
- [7] Vijay Karamcheti and Andrew A. Chien, 'A Comparison of Architectural Support for Messaging in the TMC CM-5 and the Cray T3D', *Proceedings of the 22nd Annual International Symposium on Computer Architecture*, June 1995, pp. 298-307.
- [8] Jeffrey Kuskin, David Ofelt, Mark Heinrich, John Heinelein, Richard Simoni, Kourosh Gharachorloo, John Chapin, David Nakahira, Joel Baxter, Mark Horowitz, Anoop Gupta, Mendel Rosenblum, and John Hennessy, 'The Stanford FLASH Multiprocessor', *Proceedings of the 21st Annual International Symposium on Computer Architecture*, 1994, pp. 302-313.
- [9] J. William Lee, 'Performance of User-Level Communication on Distributed Memory Multiprocessors with and Optimistic Protocol', Technical Report 93-12-06, Department of Computer Science and Engineering, University of Washington, December 1993.
- [10] NCR Corporation, *NCR 53C825 PCI-SCSI I/O Processor With Local ROM Interface Data Manual*, 1993.
- [11] Paul Pierce and Greg Regnier, 'The Paragon Implementation of the NX Message Passing Interface', Technical Report, Intel Scalable Systems Division, Beaverton, Oregon.
- [12] Rolf Riesen, 'SUNMOS Performance', Sandia National Laboratories web page linked to <http://www.cs.sandia.gov/~rolf/puma/sunmos/sunmos.html>.
- [13] Steve Sears, Michelle Dominijanni, Alan Langerman, and David Black, 'Kernel to Kernel Transport Interface for the Mach Kernel', Technical Report in OSF Research Institute Collected Papers, Operating Systems Volume 3, April 1994.
- [14] Daniel P. Siewiorek, C. Gordon Bell and Allen Newell, *Computer Structures: Principles and Examples*, New York: McGraw Hill Book Company, 1982.
- [15] Andrew S. Tanenbaum, *Modern Operating Systems*, Englewood Cliffs, New Jersey: Prentice-Hall, Inc., 1992, page 37.
- [16] Chandramohan A. Thekkath, Henry M. Levy, and Edward P. Laxowska, 'Separating Data and Control Transfer in Distributed Operating Systems', *Proceedings of the Sixth International Conference on Architectural Support for Programming Languages and Operating Systems, Operating Systems Review*, vol. 28, num. 4, December 1994, pp. 2-11.
- [17] Lewis W. Tucker and Alan Mainwaring, 'CMMRD: Active Messages on the CM-5', *Parallel Computing*, vol. 20, 1994, pp. 481-496.

- [18] Ronald J. Vetter, 'ATM Concepts, Architectures, and Protocols', *Communications of the ACM*, vol. 38, num. 2, February 1995, pp. 30-38.
- [19] Thorsten von Eicken, David E. Culler, Seth Copen Goldstein, and Klaus Erik Schauser, 'Active Messages: a Mechanism for Integrated Communication and Computation', *Proceedings of the 19th International Conference on Computer Architecture*, May 1992, pp. 256-267.
- [20] Robert Wahbe, Steven Lucco, Thomas E. Anderson, and Susan L. Graham, 'Efficient Software-Based Fault Isolation', *Proceedings of the 14th ACM Symposium on Operating Systems Principles*, December 1993, pp. 203-216.
- [21] Stephen R. Wheat, Arthur B. Maccabe, Rolf Riesen, David W. van Dresser, and T. Mack Stallcup, 'PUMA: An Operating System for Massively Parallel Systems', *Proceedings of the 27th Annual Hawaii International Conference on System Sciences*, January 1994, pp. 56-65.
- [22] Roman Zajcew, Paul Roy, David Black, Chris Peak, Paulo Guedes, Bradford Kemp, John LoVerso, Michael Leibensperger, Michael Barnett, Faramarz Rabii, and Durriya Netterwala, 'An OSF/1 Unix for Massively Parallel Multicomputers', *Proceedings of the Winter 1993 USENIX Technical Conference*, January 1993, pp. 449-468.

Randall W. Dean is a Senior Research Engineer at the OSF Research Institute where he is currently working on distributed operating system issues. Previous to joining the OSF in 1993, he spent eight years on the research staff at Carnegie Mellon University, the last four on the Mach project. His research interests include operating systems, multiprocessors, distributed computing and distributed memory management.

Author Information

David L. Black is a Research Scientist at the OSF Research institute, where he has been working on operating systems of various sorts since 1990. Previously, he was a graduate student at Carnegie Mellon University performing research on Mach, and learning how long it takes to write a Ph.D. thesis. His research interests include operating systems, multiprocessors, and real-time computing.

Randall D. Smith has been working on high-speed application messaging systems for three years, at Thinking Machines Corporation and the OSF Research Institute. He also spent three years in graduate study at the Massachusetts Institute of Technology in Computational Neuroscience, and a year working for the Free Software Foundation hacking GDB and GLD.

Steven J. Sears is currently a Senior Consulting Software Engineer with Novell Corporate Research and Development, attached to the Advanced Systems Architecture & Technologies Group. Before joining Novell he solved 'hard problems' at the OSF Research Institute. His interests include distributed operating systems and software, computer architecture, high speed networking, and distributed memory systems.

An Analysis of Process and Memory Models to Support High-Speed Networking in a UNIX Environment

BJ Murphy

Computer Laboratory, University of Cambridge, UK

S Zeadally, CJ Adams

Department of Computer Science, University of Buckingham, UK

Abstract

In order to reap the benefits of high-speed networks, the performance of the host operating system must at least match that of the underlying network. A barrier to achieving high throughput is the cost of copying data within current host architectures. We present a performance comparison of three styles of network device driver designed for a conventional monolithic UNIX kernel. Each driver performs a different number of copies. The zero-copy driver works by allowing the memory on the network adapter to be mapped directly into user address space. This maximises performance at the cost of: 1) breaking the semantics of existing network APIs such as BSD sockets and SVR4 TLI; 2) pushing responsibility for network buffer management up from the kernel into the application layer. The single-copy driver works by copying data directly between user space and adapter memory obviating the need for an intermediate copy into kernel buffers in main memory. This approach can be made transparent to existing application code but, like the zero-copy case, relies on an adapter with a generous quantity of on-board memory for buffering network data. The two-copy driver is a conventional STREAMS driver. The two-copy approach sacrifices performance for generality. We observe that the STREAMS overhead for small packets is significant. We report on the benefit of the hardware cache in ameliorating the effect of the second copy, although we note that streaming network data through the cache reduces the level of cache residency seen by the rest of the system.

A barrier to achieving low jitter is the non-deterministic nature of many operating system schedulers. We describe the implementation and report on the performance of a kernel streaming driver that allows data to be copied between a network

adapter and another I/O device without involving the process scheduler. This provides performance benefits in terms of increased throughput, increased CPU availability and reduced jitter.

1. Introduction

The integrated support of distributed continuous media systems poses design issues for the architecture of the network over which such a system is distributed and for the operating system running in the end-user terminals. This paper addresses the subject of operating system support for audio and video, particularly focussing on the reduction of delay and jitter.

The UNIX operating system was not designed to provide optimal support for continuous media. Traditionally, fair sharing of resources between multiple users has been the prime concern. However, as UNIX moves from main-frame to desktop, high throughput and deterministic response times become increasingly important.

This paper considers a number of techniques that are relevant to the support of high-speed networks in general and continuous media in particular within a conventional, and unmodified, monolithic UNIX kernel. We make two contributions. Firstly, we present the design of a network interface based on a zero-copy memory model, we compare its performance with single-copy and conventional two-copy models, and we discuss compatibility, protection and other implications. Secondly, we present the design of a process model in which data is streamed between device drivers without traversing the user-kernel boundary. Our motivation was to explore the options for integrating continuous media into a traditional operating system.

2. Architectural Background

2.1. Data Copying

The most important factor that limits application-network throughput in current generation workstations is the high cost of copying data relative to the cost of processing that data. This is because improvements in memory performance have not kept pace with improvements in processor performance [11]. Furthermore, improvements in network technology now mean that memory bandwidth is often within the same order of magnitude as network bandwidth. In order to exploit the throughput potential of the network, therefore, the number of copies between application and network adapter must be kept to a minimum.

A conventional network subsystem in a monolithic kernel such as UNIX requires two copies of the data in both the transmit and receive directions: one copy is required between application and kernel; an additional copy is required between kernel and network adapter. The network subsystem generally performs some level of protocol processing depending on the requirements of the application. For example, in order to provide a reliable byte-stream service, the network subsystem implements a communications protocol such as TCP which performs flow control and error recovery. The network subsystem manages kernel buffers according to the specification of the protocol in order to handle functions such as packet retransmission and re-assembly.

One opportunity to improve performance is to eliminate the copy from kernel to network adapter. This requires that the network adapter be equipped with on-board memory which can be mapped into host address space. In the transmit direction, the network subsystem copies data from application address space directly into adapter memory. Protocol processing may be performed and protocol headers inserted before the adapter is instructed to transmit the packet. In the receive direction, the network adapter assembles an incoming packet in adapter memory. The network subsystem may then perform protocol processing on the packet before copying the data directly into application address space. This is the approach taken by the designers of the Medusa FDDI adapter [1] and the Afterburner [3] TCP/IP

accelerator board. A further optimisation is the addition of hardware assist for on-the-fly checksum calculations as the data is copied between kernel and user space. The only changes that are required are to the network subsystem; existing applications reap the benefits of reduced copying without change.

Another opportunity to improve performance is to eliminate the copy between application and kernel. A number of virtual memory (VM) techniques have been proposed including page-remapping (in which pages are unmapped from one protection domain and mapped into the other) and copy-on-write (in which pages are shared between protection domains and copying is delayed until a process in one domain attempts to write to a shared page). These techniques are particularly relevant to micro-kernel architectures in which data may traverse multiple protection domains [5]. An alternative technique for eliminating the user-kernel copy is by the use of statically shared memory. All of these techniques normally require some degree of co-operation from the application (for example, using page-aligned buffers or pinning pages into memory) as well as possible changes to the VM and network subsystems.

By themselves, each approach (eliminating kernel-adapter copies and eliminating user-kernel copies) provides single-copy transfers between application and network. In combination, and with the appropriate hardware, zero-copy transfers are achievable.

2.2. Context Switches

Another factor that limits application-network throughput is the cost of a context switch. Normally, a hardware interrupt is generated every time a packet arrives from the network¹. On a heavily loaded system, each such interrupt will be accompanied by a context switch from the currently running user process to the user process for which the packet is intended. Context switches between user processes are expensive. The explicit overhead involves the saving of one context (register contents, memory management information, etc) and the restoration of another. However, there is an implicit cache-performance cost that, depending on the host architecture, may dominate other overheads [14].

Apart from these throughput overheads, context switches can have a serious impact on the

¹ An intelligent network adapter may re-assemble lower-level protocol data units (such as ATM cells) in order to reduce the interrupt load on the host.

timely delivery of data. Traditional UNIX systems provide no bounds on context switch latency. This is because a process running in kernel mode cannot be pre-empted. Furthermore, the conventional time-sharing scheduling policy leads to non-deterministic response times. Modern versions of UNIX (such as System V Release 4) attempt to alleviate these problems by providing a pre-emptible kernel together with real-time scheduling classes [10]. Unfortunately, early experiences of such systems have not been encouraging [15]. Applications that involve the transfer of jitter-sensitive continuous media between devices may be better served by a mechanism that completely eliminates the need for context switches.

Context switches can be avoided by exploiting kernel-level or hardware-level streaming [4]. Kernel-level streaming involves the transfer of data from source to sink over the system bus without an application process being included in the data path. Since the data passes through memory, data manipulations by the CPU are possible. Hardware-level streaming involves the transfer of data from source to sink over a private data bus or by peer-to-peer DMA over the system bus. No CPU manipulations of the data are possible. These two types of streaming contrast with the more conventional user-level streaming arrangement in which a user process is involved in the data path.

Our interest in kernel-level streaming stems from the observation that many continuous media applications involve the transfer of data between a network adapter and another device without requiring the *manipulation* of that data. For example, a server process may move video between network and disk; a client process may move audio between network and codec. In these cases, there is no requirement to touch the data beyond the presentation-layer conversions performed by the protocol stack. The fact that the data needs to cross the user-kernel boundary (twice) is an artefact of the design of the I/O sub-system. Performing the transfer in the kernel reduces the amount of copying involved and eliminates the context-switch overhead that would otherwise be incurred. This improves throughput, improves CPU availability, and reduces jitter.

3. Implementation

3.1. Copy Avoidance

Three different styles of device driver (two-copy, one-copy and zero-copy) have been implemented for the CHARISMA Asynchronous Transfer Mode (ATM) network adapter. ATM is a switching and multiplexing scheme based on the use of small fixed-size "cells" which is likely to be deployed in many high-speed networks of the future [17]. The CHARISMA ATM adapter has been designed for the EISA bus and features 1 MB of dual-ported RAM which is memory-mapped into host address space. The on-board 25 MHz T801 transputer offers an ATM Adaptation Layer 5 (AAL5) interface to the host processor. AAL5 provides a variable-length packet transfer service with error detection on top of the fixed-length cell relay service provided by ATM [17]. All communication between the two processors is performed by the manipulation of a pair of software FIFOs in shared memory and by hardware interrupts. As noted in [3, 6, 13], the use of a pair of shared memory FIFOs to pass buffer descriptors coupled with the atomicity of load and store instructions allow two processors to exchange data without the need for synchronisation primitives. Except for the small amount of space used by these FIFOs and ancillary control structures, the shared memory is organised as two pools of fixed-sized buffers (4 KB, equal to the VM page size), one pool for transmission and the other for reception. Data is transferred using memory load/store instructions. The CHARISMA host adapter is described in detail in [13].

The two-copy "atm" driver is a STREAMS [19] driver that provides both a DLPI [24] interface and a "raw" interface to the underlying AAL5 service. Within the STREAMS model, device drivers, modules and multiplexors interact by passing messages between uni-directional queues. A STREAM head provides user processes with the point of access to a particular stream (queue-pair). No copying is involved as data is passed between queues since messages are passed by reference. The DLPI interface allows the driver to be linked under the standard IP multiplexor thereby supporting the transmission of IP packets over our ATM network. The raw interface allows a user process to transmit and receive AAL5 packets using the `read()` and `write()` system calls. The `atm` driver provides two-copy transfers between application and network.

The single-copy "cpatm" driver is a non-STREAMS character driver that simply copies data between application and network. It provides a `read()`/`write()` interface to the underlying AALS service. The cpatm driver provides single-copy transfers between application and network but does not permit integration with in-kernel network protocols².

The zero-copy "mmatm" driver is a non-STREAMS driver that eliminates copies between a co-operating application and the network. It supplies a `mmap()` entry point that allows an application process to map buffers (pages) from adapter memory directly into user address space. This allows the data path between application and network to bypass the kernel entirely. The control path is managed by the mmatm driver. Before an application can transmit a packet, it must make an `ioctl()` request to the driver to obtain a buffer identifier. The buffer identifier specifies the area of adapter memory that has been allocated to the application. The application then writes its data directly into the specified buffer on the adapter. In order to transmit the data, the application makes an `ioctl()` call to the driver passing the buffer identifier as the argument. In order to receive a packet, the application makes a (potentially blocking) `ioctl()` call; when a packet arrives, the driver passes back the buffer identifier as a return parameter to this call. The application consumes the data and makes an `ioctl()` call to release the buffer. Notice that the mmatm driver does not provide `read()` or `write()` entry points. Notice also that two system calls (one for buffer management and one for data transfer) are required to transfer a packet in each direction. These issues are discussed later.

3.2. Context-Switch Avoidance

The kernel streaming "kproc" driver is a STREAMS multiplexor under which two device drivers may be linked. The kproc driver can be instructed to copy data from the read queue of one driver to the write queue of the other, and vice-versa (Figure 1). This is exactly the behaviour of a STREAMS-based IP multiplexor that has been configured to act as a router. The presence of the

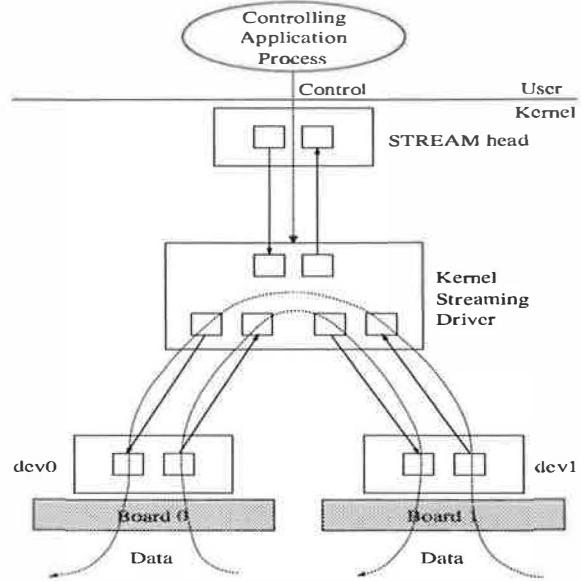


Figure 1: STREAMS-based Kernel-level Streaming Implementation

kproc driver is transparent to the devices linked below it; each driver is unaware of whether its queues are connected to a user process or to another driver. The kproc driver allows data to be transferred between a pair of devices using only two copies instead of four (since the two copies across the user-kernel boundary are eliminated) and without any context switches.

A key feature of this model is the separation of the control path from the data path [18]. The data path is within the kernel, while an application process may control the flow by means of an out-of-band channel. This is a radical departure from the conventional UNIX I/O model. Although our kproc driver does nothing more than transfer data between devices, it would be possible to perform in-band processing should that be required. For example, flow-specific events (such as video frame boundaries) could be signalled to the application in order to support higher level synchronisation services. Furthermore, additional STREAMS modules could be "pushed" onto the data path to perform higher level protocol processing such as transport-level control and presentation-level format conversions.

² A STREAMS procedure (`esballoc()`) enables a device driver to eliminate the copy from a memory-mapped adapter to a kernel buffer by allowing a STREAMS message to be wrapped around a user-supplied buffer. (One of the arguments to `esballoc()` is a pointer to a function to be called to de-allocate the buffer.) We suggest that the SVR4 STREAMS implementation be modified to provide the STREAM head with the same facility. This would permit the implementation of an integrated single-copy STREAMS-based network subsystem in both the receive and transmit directions. None of our experiments described here made use of the `esballoc()` procedure.

4. Experimental Detail

In order to evaluate the performance of these models, a simple traffic generator/receiver program was written for the T801 transputer on the network adapter. The program can be instructed (via a shared memory interface) to generate a series of packets of specified size and at a specified rate or to sink a series of packets. In combination with an equivalent program running as a UNIX user process, measurements were taken to determine the performance of each driver. Each program has access to the 1 μ s clock provided by the T801; the transputer program copies the value of the clock into shared memory (at a granularity of approximately 4 μ s) for the benefit of the UNIX program. The programs timestamp outgoing and incoming packets in local memory. At the end of a test run, the transputer program is instructed to dump its array of timestamps into shared memory. Elapsed times are calculated by simply subtracting corresponding timestamps. These values represent the time taken to transfer a series of packets between a network adapter and a UNIX user-level process.

In order to measure the performance of the kernel streaming driver, we equipped the host with two of our network adapters. Each board runs a separate copy of the transputer test program. A UNIX control program first links the two instances of the device driver below the kproc multiplexor. It then configures the transputer program on one board to act as a sink of packets and the other transputer program to act as a source of packets. At the end of the test sequence, the control program instructs each transputer to dump its array of timestamps. Elapsed times are calculated allowing for the offset between the two clocks. These values represent the time taken to transfer a series of packets between a network adapter and another hardware device using kernel-level streaming.

Two variations of kernel-level streaming were investigated: synchronous streaming, in which packets are transferred directly by the interrupt handler, and asynchronous streaming, in which packets are only queued by the interrupt handler and transferred some time later by the STREAMS scheduler (with most interrupts enabled). It is possible to configure the style of message passing (synchronous or asynchronous) on a per-stream basis by means of an `ioctl()` call.

The kernel streaming configuration just described was compared with a conventional

user-level streaming configuration. A UNIX user-level streaming program was used to read packets from one board and write packets to the other board in the traditional manner using the "raw" `read()/write()` interface provided by the STREAMS atm driver. The streaming program was run as both a time-sharing and a real-time process.

In order to gain a proper understanding of the relative merits of kernel-level streaming versus user-level streaming, we conducted these experiments under a number of different operating conditions. The first set of readings was taken without artificially loading the system. A second set of readings was taken while running a compute-intensive "soak" program comprising a tight loop. The "soak" program ensures that the processor is never idle. It spends no time in the kernel. A third set of readings was taken while running an I/O-intensive "find" program comprising a recursive search of an NFS filing system over Ethernet. The "find" program spends a considerable proportion (over 95%) of its execution time in the kernel performing system calls. Furthermore, the "find" program causes interrupts to be generated by the Ethernet adapter. It spends approximately 85% of the time sleeping (waiting for I/O). The interrupt priority level of the Ethernet driver was the same as that of our ATM driver. The "soak" and "find" processes were run in the default time-sharing scheduling class.

All our measurements were made on a 33 MHz Intel 80486 EISA machine running an otherwise unloaded UNIX System V Release 4.2 kernel in the default multi-user state. The machine features 32 MB of RAM, 8 KB of physically-indexed four-way set associative write-through primary cache, and 256 KB of physically-indexed two-way set associative write-back secondary cache.

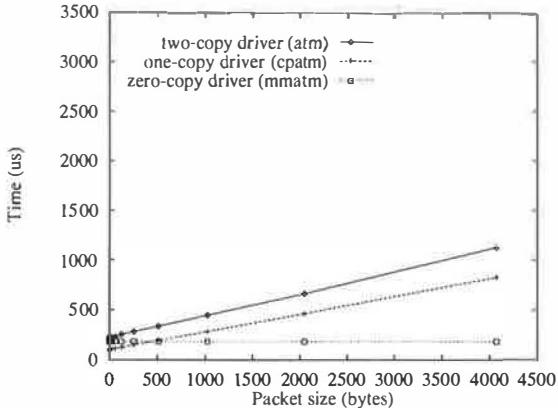


Figure 2: Driver Transmit Performance
(Caching Enabled)

5. Results

5.1. Comparison of Device Driver Performance

Figures 2 and 3 illustrate the performance of the three types of device driver for different sizes of packet. The data points represent the median of 1000 timing measurements for each size of packet using an inter-packet transmission interval of 10 ms.

5.1.1. Relative Cost of Transmission versus Reception

The first point to notice is that the cost of reception is approximately twice that of transmission for all three types of driver. This is typical of network subsystems in general. It is explained by the fact that whereas packets are transmitted synchronously with respect to the source they are received asynchronously with respect to the sink. This normally demands that the operating system handle an interrupt, wakeup the waiting process and possibly switch context for every incoming packet.

5.1.2. Relative Performance of Zero-Copy versus Single-Copy Drivers

The second point to observe is that for small packets (less than approximately 500 bytes) the performance of the single-copy cpatm driver is better than that of the zero-copy mmatm driver. This is because we have included the cost of the system call required to perform buffer management for each packet transfer in our readings for the zero-copy driver. Recall that an mmatm user wishing to transmit a packet must first ask the driver for a free buffer. Likewise, when an mmatm user has finished

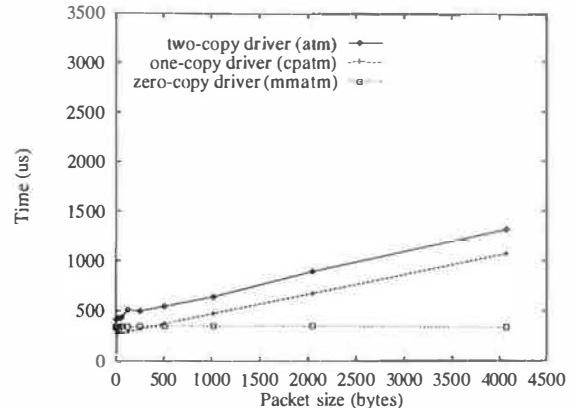


Figure 3: Driver Receive Performance
(Caching Enabled)

processing an incoming packet, it must return the buffer to the driver. This additional overhead is included in our analysis in order to permit a fair comparison.

5.1.3. Relative Cost of STREAMS versus non-STREAMS Drivers

As expected, the STREAMS atm driver provides the worst performance. There are two explanations for this. Firstly, the STREAMS driver performs two copies of the data. Secondly, there is an inherent overhead associated with STREAMS as indicated by the difference between the points of intersection on the ordinate for the STREAMS atm and the non-STREAMS cpatm drivers. That is, for very small packets for which the overhead associated with copying is negligible, the performance of the STREAMS driver is worse (by a factor of approximately 2) than that of the non-STREAMS driver. Part of the reason for this is that every packet on a stream takes the form of a STREAMS message with an associated header that requires allocation and de-allocation by the driver and STREAM head. A further reason is that the movement of a message between driver and application involves processing by the STREAM head and possibly by the STREAM scheduler in addition to the driver. The inherent STREAMS overhead becomes less significant with increasing packet size as the copying overhead becomes dominant.

5.1.4. Effects of the Cache

The final point to notice is that the performance of the two-copy driver does not degrade with packet size as quickly as one might anticipate

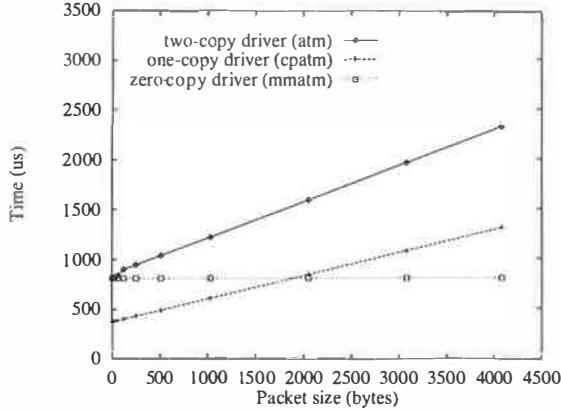


Figure 4: Driver Transmit Performance
(Caching Disabled)

given the extra copy involved relative to the single-copy driver. This is illustrated by the fact that the gradient of the lines relating to the two-copy atm driver is very nearly the same as the gradient of the lines relating to the single-copy cpatm driver. We speculated that this was due to the hardware cache, but decided to investigate further. Figures 4 and 5 show a repeat of the experiments with all caches (primary and secondary) disabled. Under these conditions, the throughput characteristics of the two-copy atm driver demonstrate the cost of the extra (un-cached) copy. Specifically, the gradient of lines relating to the two-copy atm driver is greater (by a factor of approximately 1.7) than the gradient of lines relating to the single-copy cpatm driver. With caches on (Figures 2 and 3), the effect of the copy between main memory and main memory (across the user-kernel boundary) is ameliorated by the fact that the data is already cached. This degree of cache residency is unlikely to be achievable under realistic load conditions in which manipulation of network buffers is interleaved with memory accesses from other processes. Furthermore, the benefit of the cache to a network subsystem that performs multiple copies is at the expense of reduced cache residency seen by other processes in the system. Neither of these drawbacks are revealed by our experiments.

The cost of the copy between adapter memory and main memory is almost independent of whether or not caches are enabled. This is illustrated by the fact that the gradient of the lines relating to the single-copy cpatm driver in Figures 2, 3, 4 & 5 is virtually constant. This is because the adapter memory is configured to be non-cacheable to prevent cache consistency problems [20]. The cost of accessing adapter memory over the system bus

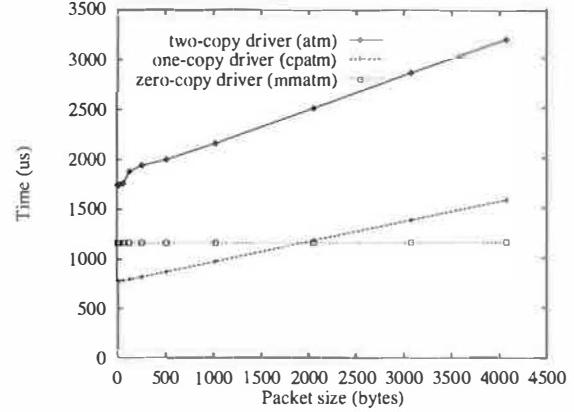


Figure 5: Driver Receive Performance
(Caching Disabled)

dominates the cost of accessing main memory regardless of whether or not caches are enabled.

5.2. Comparison of User-level Streaming versus Kernel-level Streaming

Figures 6, 7 and 8 illustrate the performance of the various streaming implementations under three operating conditions. We have adopted the graphical presentation format devised by Faller [9]. The ordinate shows the time taken to transfer a 4072-byte block (the maximum size of an AAL5 service data unit that fits into a 4 KB buffer) between one network adapter and the other. The abscissa shows the percentage cumulative frequency of these transfer times. The value on the ordinate corresponding to $P\%$ on the abscissa is called the P th percentile of the distribution. For example, an 80th percentile of 2 ms means that 80% of the measured transfer times were at or below 2 ms. The value of the 50th percentile is, by definition, the median. A flat graph (ie a horizontal line) represents zero variance. We have used a logarithmic scale on the ordinate in order to dampen the visual distortion that would otherwise be caused by a small number of large readings. We have used an exponential scale on the abscissa in order to highlight the relatively small number of readings of particular interest. Each test run comprised the transfer of 1000 packets with an inter-packet transmission time of 10 ms. The results are plotted at 1% intervals on the abscissa. The slight positive gradient that is observable in the graphs (approximately 130 μ s over the 10 s measurement period) is due to drift between the two transputer clocks.

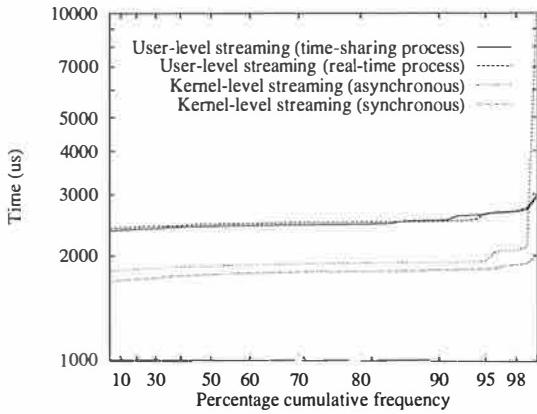


Figure 6: Streaming Performance with System Idle

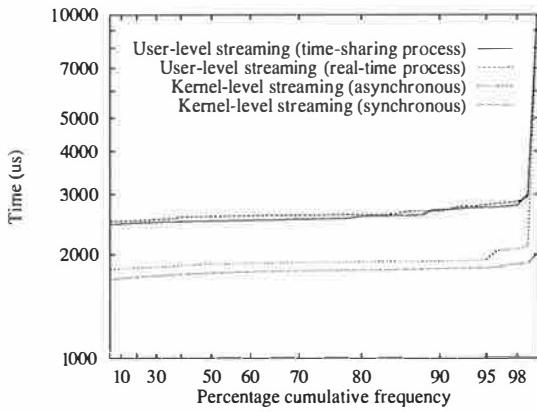


Figure 7: Streaming Performance with Compute-Intensive Load

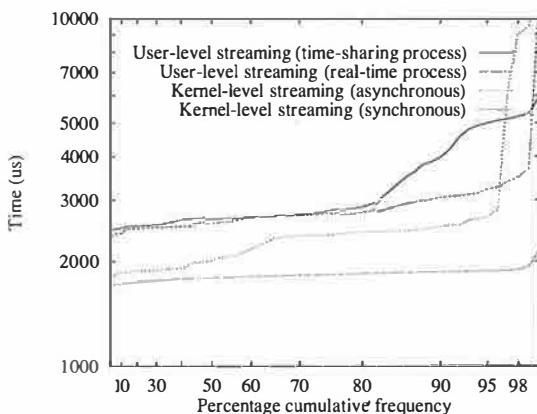


Figure 8: Streaming Performance with I/O-Intensive Load

5.2.1. Kernel-level Streaming with no Artificial Load

Figure 6 shows the performance of the four streaming implementations with no artificial load. Kernel-level streaming provides a throughput increase of approximately 25%. Furthermore, average CPU utilisation (as measured by the public domain `top` program) is reduced by between 30% and 50% depending on whether the transfer is performed by the STREAMS scheduler or by the interrupt handler respectively. A comparison of the graphs for the two variations of kernel-level streaming demonstrates the overhead imposed by the STREAMS scheduler (approximately 5%).

5.2.2. Kernel-level Streaming with Compute-Intensive Load

Figure 7 shows the minimal impact of the compute-intensive load on the two user-level streaming models. The only performance penalty is the cost of pre-empting the "soak" process for each packet transfer. This overhead (which is just discernible from a comparison of Figures 6 and 7) is approximately 90 μ s. The reason why the "soak" process does not have a more serious impact on the performance of the time-sharing user-level streaming process is that, although both are competing for the processor within the same scheduling class, the scheduler ensures that the priority of the I/O-bound process remains higher. This policy is to help provide better response times for interactive processes compared with compute-bound processes. There is no discernible reduction in the performance of the kernel-level streaming implementations.

5.2.3. Kernel-level Streaming with I/O-Intensive Load

Figure 8 shows the impact of an I/O-intensive load. The real-time user-level streaming process displays much better jitter characteristics than the time-sharing process. This is because pre-emption points within the kernel allow the real-time streaming process to pre-empt the "find" process while the latter is in the kernel. The synchronous implementation of kernel-level streaming is virtually unaffected by the extra I/O load; this is significant, but not surprising since all the work is being done in the interrupt handler. The asynchronous implementation of kernel-level streaming performs badly for approximately 5% of the data set; we are unable to explain this behaviour.

The maximum throughput of the kernel-level streaming model (approximately 18 Mbits/s), which

involves two copies of the data, is bounded by adapter memory to main memory bandwidth (approximately 40 Mbits/s). It is reasonable to speculate that higher rates would be achievable using faster memory.

6. Discussion

6.1. Copy Avoidance

The elimination of all copies (using the shared memory technique described here), whilst attractive for performance reasons, raises a number of issues that we address below.

6.1.1. Hardware Constraints

Not all network adapters offer a memory-mapped interface without which copy elimination is impossible. Many peripherals reside on an I/O bus and make use of DMA or programmed I/O for data transfer; the choice between the two is often dictated by the host bus architecture [1]. We suggest that the potential performance benefits of zero-copy transfers need to be taken into account when making the choice between the memory bus and the I/O bus during the adapter design stage and when evaluating the merits of competing host bus architectures.

An additional problem is that on-board memory is a limited resource. Buffering protocol data (in order to provide single-copy transfers) or even application data (in order to provide zero-copy transfers) in adapter memory rather than main memory may not be realistic for high volume flows. Nevertheless, where memory-mapped designs are applicable, while the discrepancy between CPU performance and memory performance remains, and while the cost of memory continues to fall, we believe that there is a strong case for furnishing the network adapter with generous quantities of RAM.

6.1.2. Application Program Interface

Current network APIs, such as BSD sockets and SVR4 TLI, allow applications to allocate arbitrarily aligned buffers located anywhere in process address space and which are contiguous in virtual (but not necessarily physical) address space. The cost of this amenity is generally an extra copy by the network subsystem thereby limiting the scope for copy avoidance. We endorse a model in which network buffers are allocated by, and at the convenience of, the network subsystem rather than the application [4]. Responsibility for buffer de-allocation is shared between the network

subsystem and the application. The transmission paradigm becomes one in which the application requests a buffer from the network subsystem, writes to the buffer, and then requests transmission. The reception paradigm becomes one in which the application requests reception of data (optionally specifying a limit on size), is handed a buffer by the network subsystem, consumes the data and de-allocates the buffer. A network buffer should be represented as an abstract data type rather than a single contiguous buffer [12]. This leaves the network subsystem free to implement such types in an optimal manner.

The advantage of such an API is its generality. Where hardware permits, it allows zero-copy transfers. Where hardware or other constraints rule out copy elimination, it facilitates single-copy transfers without having to "bend" existing APIs (by mandating page-aligned buffers, for example). We suggest that user-level protocols [7, 23] would benefit from a new API: the semantics of existing APIs are likely to force a copy even when the protocol is implemented as a user-level library and no protection domains are being crossed.

The disadvantage of such an API is that it is incompatible with current network APIs and therefore cannot be of benefit to the great many existing applications that are based on these APIs. Nevertheless, we believe that there is a new class of applications that could benefit from the performance improvements made possible by an API that is better integrated with the network subsystem. Clearly, a variety of APIs can exist side by side: continuous media applications would benefit from the new API and backward compatibility could be provided by traditional APIs at the cost of reduced performance.

6.1.3. Protection

A ramification of this new API is that responsibility for the de-allocation of buffers used by the network subsystem is pushed up from the kernel into the application layer. Furthermore, there are protection implications when such an API is underpinned by a shared memory implementation such as that described here. In order to circumvent these problems, buffers need to be managed as pools each of which is owned by a separate process. The size of each pool could be based on the throughput requirements of the application as specified at call setup time or on the dynamics of the flow subject to a system-specified per-application limit. The device driver would permit the application to map in only the

shared memory pages belonging to its pool. Thereafter, the VM subsystem ensures that an application cannot interfere with data belonging to another process. A malicious or badly programmed application that does not free its buffers only exhausts its private pool and does not compromise the operation of the entire system.

6.1.4. Operating System Integration

The zero-copy driver discussed here does not add value to the network service provided by the network adapter. Higher-level protocols could be implemented in user-space based on this design. If it is necessary to implement such protocols in the kernel, however, changes need to be made to the operating system to enable the network subsystem to handle the abstract buffer type outlined above.

6.1.5. Early Demultiplexing

A key requirement in this design is that an early demultiplexing decision can be made on incoming data [4]. In ATM networks, the VCI is ideal for this purpose provided that multiple higher-level protocols are not multiplexed over the same channel [22].

6.1.6. Performance For Small Packets

A problem with our zero-copy implementation is that, for small packets, the buffer management overhead outweighs the benefits of copy elimination. This is due to the cost of making a system call. A solution is to employ a shared memory interface between application and driver for the purpose of buffer management much like the shared memory interface between driver and adapter for the purpose of data transfer. Other researchers have noted this problem and propose a similar solution [21].

6.1.7. Cache Performance

A feature of the zero-copy model presented here is that network data is not brought into the cache unless and until it is explicitly copied by the processor. This hurts protocol stack implementations that perform multiple touches of the data since all accesses to network buffers go over the bus. However, there are a number of benefits. Firstly, the level of cache residency seen by the rest of the system increases if network data does not enter the cache. Secondly, incoming network data is only brought into the cache if and when the application consumes the data (ie as late as possible). This maximises cache

residency by eliminating the potential for context switches between the data being brought into the cache (by the network subsystem) and the data being consumed by the application [16]. Note that the performance penalty incurred by making non-cacheable accesses to adapter memory is reduced with protocols that touch only part of a packet (eg the header) rather than the entire packet. Such protocols generally sacrifice error detection (by eliminating the checksum, for example), but many "raw" multimedia flows are resilient to some degree of data corruption. Furthermore, implementation strategies based on integrated layer processing [2] ensure that the cost of accessing adapter memory is kept to a minimum.

6.2. Context-Switch Avoidance

Kernel-level streaming has the potential for delivering increased throughput, increased CPU availability, and reduced jitter. In order to realise these benefits in a generic fashion, however, the entire I/O sub-system needs restructuring. For example, in order to be able to stream data directly from disk to the network, the filing system needs to offer an appropriate interface. Nevertheless, we believe that this technique is applicable to a large class of multimedia I/O devices.

An added feature of this model is that careful implementations can avoid streaming continuous media through the cache. For example, if an adapter is equipped with on-board memory, data could be transferred from that device by copying pointers to the data (in the form of STREAMS messages wrapped around adapter buffers, for example) rather than the data itself. Alternatively, data could be transferred between adapters using DMA via main memory and, provided that protocol processing does not require touching the entire packet, the impact on the cache is minimised.

The concepts of kernel-level and hardware-level streaming are not new [4, 8, 18]. We believe that kernel-level streaming is a good compromise between hardware-level streaming and user-level streaming. Hardware-level streaming does not involve the processor: all manipulation of the data, including protocol processing, must be performed by intelligent peripherals. User-level streaming is the most flexible but incurs the highest costs in terms of performance and resource usage. We observe that, unlike other implementations [8], the STREAMS I/O sub-system

supports kernel-level streaming without requiring modifications to the operating system.

We have not addressed the issue of streaming *multiple* flows between devices within the kernel. Under such circumstances, there will be "cross-talk" between the flows. The FIFO scheduling discipline provided by our prototype implementation does not serve the needs of multiple flows with conflicting "quality of service" requirements. In passing, we note that the STREAMS scheduler supports multiple priority levels which could be exploited in a more sophisticated implementation to minimise cross-talk. Nevertheless, we believe that a better solution would be an operating system that provided a single integrated scheduling mechanism for all CPU activity including that related to inter-device flows, conventional application-terminated flows, and compute-bound tasks.

7. Conclusions

We have reported on the performance characteristics of a number of process and memory models to support high-speed networks within a monolithic UNIX kernel. We have presented the design of a zero-copy device driver for an ATM network adapter. The design uses shared memory which is mapped directly into user address space. We have demonstrated the performance benefits and discussed the implications of such a design.

We suggest that existing APIs, such as BSD sockets and SVR4 TLI, are a barrier to achieving high network throughput. We advocate a new style of API in which buffers are allocated by the network subsystem rather than by the application. We suggest that such an API improves the scope for copy avoidance.

We have demonstrated that streaming data between two devices in the kernel provides performance benefits in terms of increased throughput, increased CPU availability and reduced jitter. We are not suggesting that kernel-level streaming is the ideal mechanism to manage the resources of a multimedia workstation. Nor do we casually make the suggestion that application functionality be embedded in the kernel. Rather, we view kernel-level streaming as a pragmatic approach to improving support for continuous media within the constraints of a conventional operating system.

8. Acknowledgements

We gratefully acknowledge the contribution of our colleagues at the Rutherford Appleton Laboratory, UK, who designed and built the ATM network adapter on which our experiments were based. We would also like to thank the various reviewers who commented on earlier versions of this paper. The work was conducted at the University of Buckingham, UK, as part of the CHARISMA project and funded by the European RACE II research programme (project number R2071).

9. References

- [1] D Banks and M Prudence, "A High-Performance Network Architecture for a PA-RISC Workstation", *IEEE Journal On Selected Areas in Communications*, Vol 11, No 2, February 1993.
- [2] DD Clark and DL Tennenhouse, "Architectural Considerations for a New Generation of Protocols", *Proc ACM SIGCOMM '90*, Philadelphia, USA, September 1990.
- [3] C Dalton, G Watson, D Banks, C Calamvokis, A Edwards and J Lumley, "Afterburner", *IEEE Network*, Vol 7, No 4, pp 36-43, July 1993.
- [4] P Druschel, M Abbot, M Pagels, L Peterson, "Network Subsystem Design", *IEEE Network*, Vol 7, No 4, pp 8-17, July 1993.
- [5] P Druschel and L Peterson, 'Fbufs: A High-Bandwidth Cross-Domain Transfer Facility', *Proc 14th Symposium on Operating System Principles*, 1993.
- [6] P Druschel, L Peterson, B Davie, "Experience with a High-Speed Network Adaptor: A Software Perspective", *Proc ACM SIGCOMM '94*, London, England, September 1994.
- [7] A Edwards, G Watson, J Lumley, D Banks, C Calamvokis, C Dalton, "User-space Protocols Give High Performance to Applications on a Low-cost Gb/s LAN", *Proc ACM SIGCOMM '94*, London, England, September 1994.

- [8] K Fall and J Pasquale, "Exploiting In-Kernel Data Paths to Improve I/O Throughput and CPU Availability", *Proc USENIX Winter Technical Conference*, San Diego, CA, USA, January 1993.
- [9] Newton Faller, "Measuring the Latency Time of Real-Time Unix-like Operating Systems", <ftp://icsi.berkeley.edu/pub/tech-reports/1992/tr-92-037.ps.Z>, 1992.
- [10] Berny Goodheart and James Cox, *The Magic Garden Explained: The Internals of UNIX System V Release 4*, Prentice Hall, ISBN 013-098138-9, 1994.
- [11] JL Hennessy and DA Patterson, *Computer Architecture: A Quantitative Approach*, Morgan Kaufmann Publishers Inc, ISBN 1-55860-188-0, 1990.
- [12] NC Hutchinson and LL Peterson, "The x-Kernel: An Architecture for Implementing Network Protocols", *IEEE Transactions on Software Engineering*, Vol 17, No 1, January 1991.
- [13] BJ Murphy, CJ Adams, S Zeadally, "The CHARISMA ATM Host Interface", *Proc 3rd International Conference on Broadband Islands*, Hamburg, Germany, June 1994. Also available electronically as <ftp://ftp.cl.cam.ac.uk/users/bjm22/bris94.ps.gz>.
- [14] JC Mogul and A Borg, "The Effect of Context Switches on Cache Performance", *Fourth International Conference on Architectural Support for Programming Languages and Operating Systems*, April 1991.
- [15] J Nieh, JG Hanko, J Duane Northcutt and GA Wall, "SVR4 UNIX Scheduler Unacceptable for Multimedia Applications", *Proc 4th International Workshop on Network and Operating System Support for Digital Audio and Video*, Lancaster, UK, 1993.
- [16] MA Pagels, P Druschel and LL Peterson, "Cache and TLB Effectiveness in the Processing of Network Data", Technical Report TR 93-4, Dept of Computer Science, University of Arizona, Tucson, USA, 1993.
- [17] C Partridge, *Gigabit Networking*, Addison-Wesley Professional Computing Series, ISBN 0-201-56333-9, 1993.
- [18] J Pasquale, "I/O System Design for Intensive Multimedia I/O", *Proc IEEE Workshop on Workstation Operating Systems*, Key Biscayne, FL, USA, April 1992.
- [19] Stephen A Rago, *UNIX System V Network Programming*, Addison-Wesley Professional Computing Series, ISBN 0-201-56318-5, 1993.
- [20] Curt Schimmel, *UNIX Systems for Modern Architectures*, Addison-Wesley Professional Computing Series, ISBN 0-201-63338-8, 1994.
- [21] JM Smith and CBS Traw, "Giving Applications Access to Gb/s Networking", *IEEE Network*, Vol 7, No 4, pp 44-52, July 1993.
- [22] DL Tennenhouse, "Layered Multiplexing Considered Harmful", *Proc IFIP Workshop on Protocols for High-Speed Networks*, Zurich, Switzerland, May 1989.
- [23] C Thekkath, T Nguyen, E Moy, E Lazowska, "Implementing Network Protocols at User Level", *Proc ACM SIGCOMM '93*, San Francisco, USA, September 1993.
- [24] UNIX International OSI Special Interest Group, *Data Link Provider Interface Specification*, UNIX International, Waterview Corporate Center, 20 Waterview Boulevard, Parsippany, NJ 07054, USA, Revision 2.0.0, August 1991.

Author Information

Brendan Murphy is a post-doctoral Research Associate at the University of Cambridge currently working on the integration of continuous media "streams" into a CORBA environment. His other research interests include network protocol design and operating system support for high-speed networks.

Sherali Zeadally is finishing his PhD in Computer Science at the University of Buckingham. His research interests include operating system support for multimedia, I/O sub-system design,

especially to support high-speed networks, and distributed systems.

Chris Adams is Professor of Computer Science at the University of Buckingham and a member of the Advanced Communications Unit at the Rutherford Appleton Laboratory. His research interests range from global network strategies to hacking processor microcode. They include satellite communications, multimedia applications, multiservice networks and operating systems.

The authors can be reached via electronic mail to `charisma@buck.ac.uk`. An electronic version of this paper is available as `ftp://ftp.cl.cam.ac.uk/users/bjm22/usenix96.ps.gz`. Further information on the CHARISMA project can be found via `http://www.brookes.ac.uk/cms/research/dsproj.html#CHARISMA`.

Zero-Copy TCP in Solaris

*Hsiao-keng Jerry Chu
SunSoft Inc.*

Abstract

This paper describes a new feature in Solaris that uses virtual memory remapping combined with checksumming support from the networking hardware, to eliminate data-touching overhead from the TCP/IP protocol stack. By implementing page remapping operations at the right level of the operating system, and caching MMU mappings to take advantage of locality of reference, significant performance gain is attained on certain hardware platforms. Nevertheless, the performance improvement over CPU copying varies, depending on the host memory cache architecture, MMU design, and application behavior. We begin by comparing different zero-copy schemes, and explain our preference for page remapping and copy-on-write (COW) techniques. We then describe our implementation, and present its performance characteristics under a number of different parameters. We conclude with ideas for future improvements.

1. Introduction

High data throughput is one of the major requirements for applications such as multimedia and video-on-demand operating over high-speed networks like ATM. Networking software or hardware is often a major bottleneck in this respect, and considerable research focuses on delivering the vast bandwidth effortlessly.

Analysis of components of networking software reveals that data copy and checksum overhead dominates processing time for high throughput applications [3, 10, 17]. Older generation networking software and hardware often required multiple data copy and separate data checksum operations on each byte of a data packet. The past few years have seen a number of successful implementations (such as in Solaris release 2.4) introducing “single-copy” (CPU copy). They are often combined with the TCP checksum calculation in one single loop, resulting in substantial throughput improvement[17].

Recent efforts focus on designing an optimal architecture capable of moving data between application domains and network interfaces without CPU intervention, in effect achieving “zero-copy*”. Issues that arise are architecture efficiency, cost, application and driver programming interfaces, software complexity, and compatibility. Proposals attaining the best efficiency often deviate from the existing UNIX networking I/O interface, or may rely on special hardware. As developers of a widely used operating system, we were resolved not to alter any interface, either at application or device driver level[†]. There is a considerable installed base of software investment to protect. Further, for broader market appeal we felt it unwise to rely on specialized hardware, as some of the cited examples in literature did.

We turned to virtual memory remapping and copy-on-write technique, both being widely adopted in operating system design to avoid copying [1, 2, 8, 16]. Although these operations are not without expense, they often can be applied transparently, and so fit our goal of minimizing software module change. Networking adaptors used are Sun’s SBus-based ATM interface cards, capable of both 155 and 622 Mb/s, OC-3 and OC-12 respectively.

In the next section, we illustrate the importance of reducing data touching overhead as packet size increases in new generations of network media. In Section 3 we introduce zero-copy, and describe several different approaches. Our implementation based on page remapping and copy-on-write is described in detail in Section 4. Section 5 examines aspects of host architectures and their impact on the performance of zero-copy and single-copy, to set the stage for the measurement results presented in Section 6. Section 7 presents our conclusions.

*. This is from the CPU point of view. A bus centric view would count DMA transfer as one copy.

†. Some additions to the driver interface were required to support hardware checksum though.

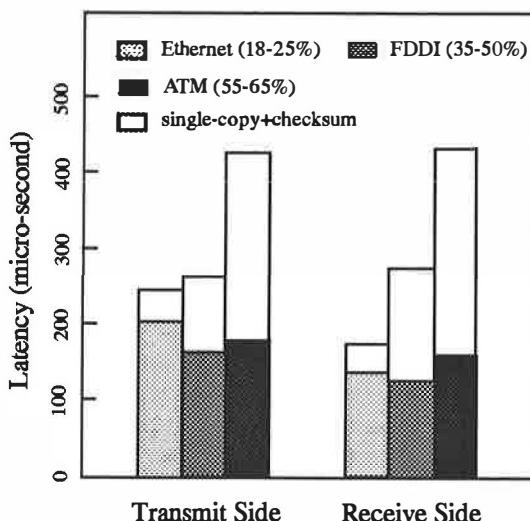
2. Networking Software Overhead

The overhead in networking software can be broken up into per-packet and per-byte costs. Generally, the per-packet cost is roughly constant for a given network protocol, regardless of the packet size, whereas the per-byte cost is determined by data copying and checksumming overhead. To reduce per-byte cost, Solaris 2.4 introduced combined single copy and data checksum design.

High throughput applications often send large packets to amortize per-packet costs over many data. However, the communication link imposes an upper bound on the packet size that can be accepted. This limit (called maximum transmission unit or **MTU**) is relatively small for traditional network media (1500 bytes for Ethernet). Therefore, the communication overhead on those networks is still dominated by the per-packet cost.

Newer generation of communication media employ larger MTUs – it is 4352 bytes for FDDI, 9180 bytes for IP over ATM, and up to 64KB for other protocols over ATM. The per-byte cost on these networks adds up to a significant portion of the total networking overhead, as demonstrated by the following chart.

Figure 1: Networking overhead in Solaris 2.5.



The figure shows per-byte (single-copy+checksum) costs as a percentage of total networking software costs for a MTU size packet, measured on SPARCstation/20 running memory-to-memory TCP tests over three different types of network.

For TCP/IP packets going through ATM networks, the copy+checksum overhead accounts for over 60% of the total networking software overhead on the end

hosts. Our goal is to further reduce the per-byte cost from the current single-copy+checksum design.

3. Zero Copy + Hardware Checksum

This section offers some insight on requirements involved in avoiding copy and checksum in networking software. We also evaluate various approaches seen in the literature.

3.1. Disk I/O versus Network I/O

Zero-copy I/O already exists in UNIX – neither memory mapped file nor raw disk I/O require any memory copy. The former offers an efficient way of accessing static, mappable objects, but is not applicable to a pipe-like network data flow. The difference between raw disk I/O and network I/O is more subtle, yet substantial. The former is synchronous in nature, while the latter is highly asynchronous.

We first discuss the write side. A raw disk write is initiated synchronously in the same program context that issues the write request. This implies the user buffer can be accessed directly by the disk driver.

The write request will block until the disk controller finishes the data transfer, which usually takes only milliseconds. Nevertheless, network I/O may take longer to really “complete” because packets may get lost somewhere in the network.

To guarantee data delivery to the destination host, a reliable data delivery protocol such as TCP must retain user data for possible retransmission. UNIX network software handles this by copying user data into a kernel buffer, and queueing it to the right outbound queue. Then it returns to the user program immediately without waiting for data transmission and acknowledgment to finish. This behavior is important for achieving high data throughput, since an application can post multiple data packets simultaneously to fill a long network pipe. Another implication of these copy semantics is that an application may safely reuse the same buffer to prepare data for the next transfer.

To achieve zero-copy, as in raw disk I/O, virtual memory (**VM**) operations can be used to lock down user memory and map it into kernel address space. This makes user data accessible to the protocol software and the device driver, even after the user program returns. But if the application reuses the same buffer too soon, it may destroy previous data still waiting to be sent, unless measures are taken to protect the user buffer before the transport layer is finished with it.

For a raw disk read, the read buffer is posted first, followed by disk I/O. Therefore, before the disk controller starts the data transfer, it already knows where to put the data.

In the networking case, packets arrive asynchronously at the network interface. There may or may not be read buffers posted from the receiving clients. Limited interface memory requires newly arrived data to be transferred into regular memory quickly to free up resources. Even if read buffers are posted, special preview of packet headers by the hardware is required to determine which read buffer to place the packet, as there are often multiple clients, each has a different read buffer.

3.2. Different Zero Copy Schemes

Several zero-copy schemes have been proposed in the literature. Smith et al. [13] outlines a variety of approaches to host interface design and supporting software. Steenkiste [14] presents a taxonomy of host interfaces and their numbers and types of data movement across a memory bus. In the following, we classify all the approaches into four categories, and briefly describe the pros and cons for each.

3.2.1 User accessible interface memory

The best scenario with minimal data transfer overhead is one in which the network interface memory is accessible and pre-mapped into user and (possibly) kernel address space. Data never needs to traverse the memory bus until is accessed by the application.

Unfortunately, this requires complicated hardware support and substantial software changes. For one thing, cache consistency has to be maintained either through software flushing or special hardware arrangement. On the receive side, to avoid remapping memory, it requires intelligence in the network hardware to direct incoming data to the right interface memory pool, since the interface memory is likely to be shared among multiple clients.

Obviously, this scheme also requires applications to use special buffer management calls to allocate and use the interface memory. Software compatibility and portability do not exist.

Limited interface memory could pose a serious resource problem. Memory hogs or bug-ridden applications with memory leaks can easily deplete the interface memory available for use [15].

A variant of this approach that uses a dedicated co-processor for protocol processing is described by Cooper et al [4].

3.2.2 Kernel–network shared memory

To alleviate the resource problem described above, this scheme lets the operating system kernel manage the interface memory, and uses direct memory access (DMA), or program I/O (PIO, i.e. CPU copy), to move data between interface memory and application buffers. Data only travels across the memory bus once (DMA) or twice (PIO), so overhead is minimized. A further advantage lies in existing applications not being required to undergo modifications, since the socket's copy semantic is fully maintained by this approach. *Afterburner*, a classic example in this category, is described by Dalton et al. in [5].

The software to support this scheme can be complicated. Kernel networking buffer management code must be enhanced to support this special pool of memory from the network interface, which it co-manages with the device driver. With TCP checksum being performed during DMA or PIO, it poses a problem on the receive side. Packets cannot be verified and acknowledged by TCP until receiving clients run and post read requests. If this doesn't happen in time, the transmit side will time-out and generate unnecessary retransmissions.

If DMA is used, it requires pinning and unpinning of user pages, and possibly mapping from the kernel context, which add to the costs [11].

Applications can no longer hog the interface memory directly, but TCP retransmit buffers still reside in the interface memory. Even though flow control exerted by TCP and the socket layer together impose an upper limit on the amount of memory each connection may consume, a few hung TCP connections can still starve interface memory.

3.2.3 User–kernel shared memory

This scheme defines a new set of application programming interfaces (APIs) with shared semantics between the user and kernel address spaces, and uses DMA to move data between the shared memory and the network interface. One proposal in this category is called *fast buffers (fbufs)* described by Druschel and Peterson in [6]. It uses a per-process buffer pool that is pre-mapped in both the user and kernel address spaces, thus eliminating the user–kernel data copy. Ideally, each data byte crosses the memory bus

exactly once, so overhead is low and no special, processor-addressable interface memory is needed.

One major disadvantage of this approach is application compatibility. All the programs have to be converted to use this alternate set of APIs and programming model.

Managing the shared buffer pool requires close cooperation between the application, networking software, and the device driver, all allocating memory from the same pool. On the receive side, virtual address and physical memory fragmentation may develop, either because the application needs to hold on to some data while relinquishing others, or when TCP handles out-of-order or duplicate packets, or when the driver passes up partially filled buffers due to smaller-than-MTU packets arriving from the network. Fragmentation not only makes the application programming model more complicated, but also may pose problems to DMA engines, as many of them have special size or alignment requirement. For the latter, it may be necessary to resort to copying.

On the receive side, network hardware must be capable of targeting DMA transfer of an incoming packet to the correct memory pool allocated by the receiving client.

Due to the nature of shared memory, an error-prone application may inadvertently modify memory contents previously sent, causing data corruption problems that are hard to debug. The shared memory model also makes it difficult to connect with other types of memory objects, as close cooperation is required of other memory managers.

3.2.4 User–kernel page remapping+COW

This scheme uses DMA to transfer data between interface memory and kernel buffers, and remaps buffers by editing the MMU table to give the appearance of data transfer. By combining it with COW (copy-on-write) technique on the transmit side, it preserves the copy semantics of traditional socket interface. It also connects well with the rest of the VM system and other types of memory objects.

This approach is not without shortcomings, however. The VM operations can be expensive. All the buffers involved must align on page boundaries, and occupy an integral number of MMU pages. On the transmit side, this is less of a problem, as any fragmented part of a user buffer can be transmitted separately, using CPU copy. On the receive side, this can be problematical. First, packets must be large enough to cover a

whole page of data. This limits the scheme to networks with an MTU size larger than the system page size, such as FDDI and ATM. Secondly, in order to apply page remapping to a stream of packets without disruption, user payload of all the packets must contain an integral number of pages. This can be accomplished by TCP negotiating a maximum segment size (MSS) of such a size. Thirdly, network drivers must arrange receive buffers in such a way that, after DMA, user payload shows up on a page boundary in the buffer. Without trailer encapsulation, driver software has to accurately predict the size of protocol headers to skip. For protocol headers of a fixed length, such as TCP/IP (assuming no options), this is easy. But for NFS and similar protocols where the header length varies depending on requests, one is forced to choose a size that works for a majority of cases. One solution requires adding special hardware logic to the DMA engine to parse packet headers, so that it knows where to split the data from the headers on the fly.

On the transmit side, even though data is write-protected from the application during transfer, applications still should avoid reusing busy buffers because copy-on-write faults can be very expensive. Without an end-to-end level acknowledgment, applications can not tell whether a buffer has been released by the transport or not. Fortunately, the networking software imposes an upper bound on how much data can be outstanding before flow control is exerted. For example, in the current TCP implementation in Solaris, the maximum socket send buffer size plus TCP window size totals to 128K. Therefore, if an application uses a circular list of buffers with total size greater than 128K, it will block before it has a chance to overwrite any busy buffer.

Once the alignment requirement is met, most of the software work is confined to the VM system. Very little change is needed in the networking code, except at the socket layer to replace *copyin()* with *mapin+cow*, and *copyout()* with *remap*. This scheme then becomes most attractive for us.

3.3. Hardware Checksum

All the approaches described above, with the exception of PIO, call upon special hardware to calculate data checksums, usually during DMA transfers. This is particularly important for zero-copy, as the cost of CPU checksum can become more conspicuous on a cold cache resulted from zero-copy. For PIO (single-copy), hardware checksum is less critical, since folding checksum calculation into the copy loop often makes the cost of software checksum negligible [17].

For protocols that store the checksum in the header, (e.g. TCP), it poses a problem on the transmit side for interface memory that is organized as a FIFO. In that case, a separate checksum pass over the data is required, unless a trailer checksum is used.

Note that pushing the data checksum calculation to the interface hardware weakens its protection against data corruption. Specifically, data corruption over I/O bus will not be caught. However, the extremely low error rate in a modern I/O bus, as well as other data transfers over the I/O bus for common devices like disks, are routinely assumed to be correct and not checked in software. We therefore conclude that such a reduction in protection is not unreasonable.

4. Design and Implementation

In this section, we outline our design goals, constraints, and briefly discuss our implementation.

4.1. Design Goals

- Fit the design into Solaris VM architecture [9, 12]. Build the new remapping + copy-on-write functionality on top of the existing VM base.
- Implement the new functionality outside of VM segment layer and vnode layer. Thus it can work transparently on any type of memory objects applications choose to use. In particular, it should support memory mapped files as well as anonymous memory.
- Minimize changes to the existing VM code to minimize impact on the current system's stability and performance.

4.2. Constraints

Constraints involved in pursuing our goals were:

- Avoid creating new interfaces, or altering the existing ones. This includes both the application level and the device driver level.
- Existing applications and drivers should benefit with minimal changes (e.g. buffer alignment).

4.3. Implementation

The key to attaining efficiency in the implementation is to take advantage of information already loaded in the MMU tables. For a user buffer in an address space, the physical page behind it and its protection mode can be quickly retrieved from the MMU if the buffer is currently mapped. This is much faster than going through the VM system using the regular top-down path. Also two new HAT[‡] layer operations,

hat_softlock() for the transmit side, and *hat_pageflip()* for the receive side, were invented to perform all the MMU operations in one call. This is much faster than calling multiple existing HAT operations to achieve the same task.

Another key to code efficiency is to focus on the fast path where the majority of cases execute, and fall back to copy for the rest. For example, the code acquires locks and manipulates VM data structures in a way that is most efficient for what it tries to accomplish. But in some cases it violates the locking order defined by the VM system. In order to avoid deadlocks, when the code discovers that a lock it tries to acquire is held already, instead of waiting for the lock to be released, it simply backs off and takes the copy path.

On the transmit side, when entering the write system call, the user buffer has to be mapped into the kernel address space. Since an application often allocates and reuses the same buffer, a per-process cache is created to keep around kernel mappings created. If the whole user buffer fits into the kernel mapping cache, after the first pass, no more map-in operation is needed. This also helps the performance considerably.

4.4. Operation Steps

In the following, we step through the major operations on both the transmit and the receive side. With the exception of the transport calling back when a packet is acknowledged, all operations are performed at the Stream head.^{**}

4.4.1 Transmit side

When entering the write system call -

1. Call *hat_softlock()* to locate the page behind a user buffer and lock it down.
2. Check into the kernel mapping cache to see if there is a cached kernel mapping to the same user page from previous requests. If so, simply reuse its kernel address. Otherwise, map in the user page.
3. Change the user protection to read-only. Mark the user page as copy-on-write page.

When the transport is finished with the user page -

1. Unlock the page.

[‡]. Hardware Address Translation. See [9].

^{**}. Solaris networking subsystem uses AT&T STREAMS framework.

2. Restore the user protection back to read-write. Tear down the copy-on-write state.
3. Do NOT unmap and deallocate the MMU resources of the kernel mapping. Cache it for possible future use.

4.4.2 Receive side

Inside the read system call -

1. Given a user buffer and a kernel buffer, call `hat_pageflip()` to look up and hold exclusive locks on both pages behind the addresses. Then flip two sets of MMU translations all in one function.
2. Fix all the VM data structures to reflect new page identities and vnode associations (*page renaming*).
3. Unlock the new data page. Give the old user page back to the driver.

5. Performance Evaluation

How well zero-copy performs against single-copy is mainly determined by how efficiently page remapping and COW operations can be made relative to memory copy. Both are heavily dependent upon the underlying machine architecture. In this section, we first examine the aspects of the host architecture that affect their operation efficiency, and describe architecture details of various Sun workstations we use. We then describe our software and hardware setup for performance measurements.

5.1. Page Remapping+COW Overhead

Most software overhead is incurred in the VM system, such as looking up memory pages and updating data structures. More significant is hardware-related overhead, such as reprogramming the MMU and flushes to keeping various caches consistent.

To remap each page, the software must

- flush the obsolete entry from the local translation look-aside buffer (**TLB**).
- flush stale data from the local data cache if the cache is virtually addressed. Note that for a virtual-indexed physical-tagged (**VIPT**) cache, such as the one used in HyperSPARC, flushes can be deferred if the new mapping has the same virtual color as the old one.
- On multiprocessor (MP) machines, flushes need to be repeated on remote CPUs. Unless special hardware is employed, software cross-call is often used to interrupt remote CPUs to perform the flush.

To set up a copy-on-write protection, one needs to

- change the MMU protection to disallow write, and flush the old TLB entry.
- flush dirty data from the local cache if the cache is virtually addressed^{††}.
- on MP machines, repeat the above flushes on remote CPUs.

To tear down copy-on-write protection, one needs to

- restore the original user protection mode in the MMU, and flush TLB entry again if necessary.

All the cache and TLB flushes above may inflict some penalty upon the applications that try to access the data later. The table below lists some hardware characteristics of the machines we use for performance measurement.

Table 1. Host Architectures

	Processor	E-cache (all copy-back)	Remote tlb flush
SPARCstation/20	60MHz SuperSPARC	1MB physical address	software xcall
SPARCstation/20-HS11	100MHz HyperSPARC	256KB VIPT	software xcall
SPARCserver-1000E	60MHz SuperSPARC	1MB physical address	XBus broadcast

5.2. CPU Copy Overhead

Modern computer architectures use sophisticated hardware caches to speed up CPU access to memory, based on the principle of locality of reference. Moreover, some advanced workstations employ sophisticated hardware to move data between different memory locations without CPU intervention. The latter has a few different variations with respect to keeping the cache consistent with the main memory, which may affect application performance. For example, the latest UltraSPARC processor offers special block load and store instructions that do not allocate in the E-cache on a miss. Therefore, block copy will NOT produce hot cache data on destination unless destination lines reside in the cache before copy.

For memory copy using regular load/store instructions, the speed is determined by the cache state of the data, both source and destination, and the effectiveness of the memory cache hierarchy. With each level

^{††}. Strictly speaking, the flush is not needed for a VIPT cache. It is currently done to simplify cache aliases handling.

of cache misses, access latency to the next level increases dramatically. A potential benefit of software memory copy is that it produces a warmer cache if data is consumed soon after the copy. On the other hand, excessive copy operations may cause cache thrashing, and impair overall system performance.

The following table shows CPU copy latencies on uniprocessor (**UP**) SPARCstation/20 and 8-way SPARCserver/1000E. Both machines use 60MHz SuperSPARC processors with small on-chip cache, and large (1MB) second level external cache. Note that the E-cache controller (MXCC) offers hardware block copy function. But it is only used in kernel-to-kernel memory copy, and is not applicable here.

Table 2. CPU copy latency (micro-second)

Copy 8192 bytes	SS20	SS1000E
Ehot+Dhot	37	37
Ehot+Dcold	62	62
Ecold+Dcold	257	345
Edirty+Dcold	330	388

The cache states on the first column apply to both the source and the destination cache lines. For example, the last row shows the latency numbers when both the source and destination have cache misses, and the target cache lines are dirty. In this case, the dirty data needs to be flushed back to main memory before the new data can be fetched. This explains why it takes longer than the previous row.

The cache hit numbers are the same on two machines, as both use the same type of processor and E-cache. The last two rows show that SPARCstation/20's memory system is faster than SPARCserver/1000E's.

5.3. DMA Architecture

Sun's SBus adopts a virtual address based DMA (DVMA) design, where a DMA address is first translated through an I/O memory management unit (**IOMMU**) before used. Driver software on the host is responsible for setting up IOMMU mappings before starting a DMA transfer.

In most of the recent Sun workstations, main memory cache coherency with DMA I/O is maintained automatically by the SBus to memory bus interface logic. Therefore, no CPU flush is needed. On machines with virtual address caches (e.g. SPARCstation/20-HS11), additional cache alias rule is required. That is, the DVMA address and the corresponding host memory address must map to the same cache lines. Otherwise, explicit cache flushes by the processor are required.

All the memory buses employ a write invalidate protocol for maintaining cache coherency. This has performance ramifications on the receive side—after a DMA transfer, the data will not be in the cache. When the processor accesses the data later, the data must be loaded from main memory into the cache first.

5.4. ATM Host Adaptors

We use Sun's SBus ATM interface card “*SunATM adaptor 2.0*” which supports both 155 and 622 Mb/s ATM interfaces, with a packet level interface to the host that hides the details of ATM cells. DVMA is used to move data between host memory and interface FIFO buffers, with a per-receive channel programmable header size for header/data splitting.

On the transmit side, TCP checksum calculation is done on the fly during DMA data transfer from host memory to interface memory. An entire packet is held in the interface buffer before segmentation, so that the checksum result can be stuffed into the TCP header. On the receive side, checksum calculation is done during cell transfers to receive buffer memory.

5.5. Network Environment

- **TCP congestion window** - It is set to the maximum of 65535 bytes to reflect a fatter network pipe based on ATM.
- **MSS** - The default MTU size for IP over ATM is 9180. TCP will negotiate a MSS option of 8K bytes when zero-copy is activated.

5.6. Benchmark Programs

Two programs were used for performance evaluation. Both require small changes to page-align their internal buffers.

- **ttcp** - This is a popular benchmark program for measuring TCP throughput. In our measurement, 16KB write buffer and 64KB read buffer are used. For zero-copy on the transmit side, 144K bytes are allocated and divided into nine circular buffers, each of 16KB in size. This ensures that no COW fault occurs (see section 3.2.4). For each test run, 90000x16K bytes of data are transferred. Both the socket send buffer and receive buffer sizes are set to 64K. The latter effectively sets the size of TCP receive window to 56K^{‡‡}.

^{‡‡}. This is the maximum TCP window size (without window scale option) rounded down to an integral number of MSS (8K).

- **ftp** - This is one of the most popular network applications. It currently uses a single 8K buffer on both the transmit and the receive sides. Besides aligning the internal buffers, the transmit side is changed to memory-map (`mmap(2)`) the source file, and use the mapped address directly to write to the network socket. This saves one memory copy from the current implementation, which simply reads data from the file to the internal buffer 8K at a time, then writes the buffer out to the network. The same technique can be used on the receive side, by reading data from the network directly into a memory-mapped file address. Combined with zero-copy, this cuts the number of copy operations from two to zero. But it requires that the target file already exists, and the file size remains unchanged. This is due to the limitation of `mmap(2)` —it cannot create, grow, or shrink files. For measurement purposes, this technique is not used. Instead, a 1MB file is `ftp'ed` into `/dev/null` repeatedly to avoid any file system operations.

5.7. Performance Metric

Comparisons are made between programs running under zero-copy mode and the same programs running under single-copy mode. Three different aspects of performance characteristics are examined -

- **TCP throughput** - The total amount of data transferred is divided by the elapsed time. Numbers are shown in million-bit per second.
- **Code latency** - The on-board micro-second resolution timer is used to compare code latency between zero-copy and single copy. Numbers are shown for operations on 8192 byte buffers (two MMU pages). For example, the number for single-copy is the time it takes to `copyin()`/`copyout()` 8KB data. Note that, in all test cases on the receive side, each user page has only ONE mapping to it. This is the majority case for the receive side zero-copy. If a user page is being mapped into multiple address spaces, the cost of changing all the mappings will certainly be higher.
- **CPU utilization** - a Sun internally developed tool `statit` similar to `vmstat` is used to gather CPU utilization of the whole system during the test run.

The E-cache controller for SuperSPARC has a special register to track cache misses. This is also a useful performance indicator that is highly correlated with performance metric above. The numbers we show include both the read and write misses.

6. Measurement Results

As we have discussed before, the performance of zero-copy and single-copy depends heavily on the host architecture. Our measurement results on different types of Sun workstations concurs with this. An application's behavior also plays an important role on performance, as we illustrate by using a number of different tests shown below.

6.1. SPARCstation/20 ttcp Test

The following table shows measurement results on SPARCstation/20, both uniprocessor and dual-processor configurations. The other end is connected to a SPARCserver/1000E. Since the latter is a faster machine, performance is bounded by CPU time (<2% idle time) on the SPARCstation/20 end in all test cases. Therefore, throughput numbers can be used as a gauge for overall performance. All numbers shown are obtained by running `ttcp` over 90000x16K bytes of data (see section 5.6).

Table 3. SPARCstation/20 ttcp Test

	Thruput (Mb/s)	Latency (μsec)	CPU (sec) usr+sys	Ecache miss %
UP-T-0	255	115	0+46	3.6
UP-T-1	180	220	1+65	8.2
MP-T-0	220	150	0+55	6.6
MP-T-1	180	220	2+66	9.1
UP-R-0	265	110	0+45	3.3
UP-R-1	175	270	1+66	10
MP-R-0	235	170	0+50	8.2
MP-R-1	185	270	0+64	16
UP-T-0-touch	150	89	32+47	4.9
UP-T-1-touch	135	175	30+56	5.6
UP-R-0-touch	150	60	37+43	8.4
UP-R-1-touch	120	210	32+67	12

T - transmit

0 - zero copy

UP - uniprocessor

R - receive

1 - single copy

MP - multiprocessor

For uniprocessor, zero-copy throughput improves by 42 and 51 percent on transmit and receive sides respectively. The table also exhibits close correlations between latency and E-cache miss rate. Note that due to the write-invalidate bus protocol (see section 5.3), on the receive side, data is completely out of the cache when copy starts. This explains the high E-cache miss rates.

For the MP configuration, expensive software cross calls are used to purge stale TLB entries from remote processors (see section 5.1). On the receive side, it takes four cross-calls to remap two 8K buffers, averaging 15 μ s ((170-110)/4) per call. The transmit side cost is smaller since two of the four MMU updates are for upgrading user protections from read-only to read-write. Their TLB flushes can be deferred.

6.2. SPARCstation/20-HS11 ttcp Test

HyperSPARC uses a virtually-indexed physically-tagged direct-mapped cache. The cache size is 256KB. When a page is remapped or its protection changed, not only the TLB, but also the cache may need to be flushed. One exception is on the receive side if a page is being remapped into a new address with the same virtual color as the old one, the cache flushing can be deferred. With a MMU page size of 4KB, there are 64 colors (256/4). The chance of avoiding cache flush during a remap is low.

Cache flushing makes page remapping and COW operations more expensive. For the transmit side MP case, zero-copy code takes even longer than single-copy code. But copying seems to cause sufficient negative side effects, possibly due to a small cache size, so that zero-copy still takes less total system time than single-copy.

Many tests here exhibited some idle time. The reason is not clear yet. This is why some throughput numbers are low.

Table 4. SPARCstation/20-HS11 ttcp Test

	Thruput (Mb/s)	Latency (μ sec)	CPU(sec) sys+idle
UP-T-0	200	150	58+0
UP-T-1	150	200	75+4
MP-T-0	165	215	67+4
MP-T-1	130	190	76+15
UP-R-0	220	140	52+2
UP-R-1	160	260	72+1
MP-R-0	190	190	64-2
MP-R-1	160	250	77-3

6.3. SPARCserver/1000E/8-way ttcp Test

In this test, throughput numbers no longer accurately reflect actual performance due to idle time caused by slower machine on the other end. Instead, total CPU time should be used as the gauge for performance. We also show per packet processing time, obtained by

dividing total CPU time over the number of packets transferred

On the transmit side, zero-copy code takes even longer to run. But the total system time is marginally less than single-copy's. The copy time of 115 μ secs for 8KB implies that the source and destination buffers are relatively warm in the E-cache (see table 2). This is also confirmed by the low E-cache miss rate.

Zero copy performs much better on the receive side, with nearly 30% reduction on total per-packet processing time. Again, source data is cold in the cache, causing the copy time (175 μ s) to be much higher than that of the transmit side. Moreover, TLB shoot-down is now handled in hardware by bus broadcast. Therefore, the software overhead is less than that of SPARCstation/20 (150 ~ 170 μ s).

Table 5. SPARCserver/1000E ttcp Test

	Processing time (μ s)	Code Latency	CPU(sec) sys	E-cache miss %
T-0	235	140	42	8.8
T-1	265	115	47.5	6.2
R-0	185	85	33	10
R-1	255	175	45	14

6.4. FTP Test

We ran our tests between a SPARCstation/20 and SPARCserver/1000E and collected data on the former. Since the SPARCserver/1000E is faster, CPU time is 100% utilized on the SPARCstation/20.

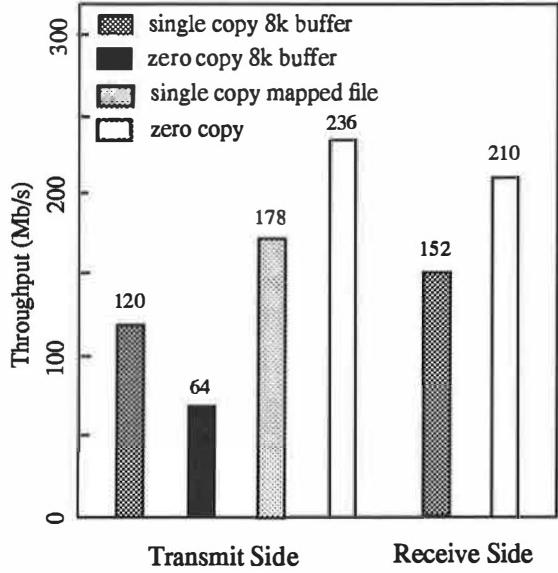
The transmit throughput nearly doubles compared to the current two-copy scheme – first from the source file page to a user buffer, then from the user buffer to a network buffer. The saving of one memory copy by using memory mapped file address accounts for about half of the performance gain.

Normally the kernel will not enable zero-copy unless the total size of “working” write buffers exceeds certain threshold. This is to avoid excessive COW faults described before. To illustrate how much impact excessive COW faults may have on zero-copy performance, the internal COW-fault-avoidance threshold is reduced to only 8K, and the 8K FTP write buffer is used. The result is a almost 50% throughput drop.

The receive side gets nearly 40% throughput boost. In a real application where incoming data is directed to a file, data access penalty due to a cold cache may cut into the gain (see next section). The best arrangement is to connect the zero-copy network to another zero-

copy I/O path, such as memory mapped files. In that case, performance gain similar to the magnitude seen on the transmit side may be realized on the receive side.

Figure 2: FTP Test on SPARCstation/20-UP



6.5. Data Touching Penalty

All the test cases so far move data from one node to another without manipulation. In reality, applications often write into the data before transmitting, and read from the data after receiving. Touching the data may cause zero-copy's performance edge over single-copy to diminish. Specifically, a warmer cache after data is touched on the transmit side will speed up single-copy, and a colder cache after zero-copy on the receive side will slow down applications accessing the data. To measure the impact, ttcp is modified to write into every word of the write buffer before transmit, and to read from every word of the data after it is received. The result is shown in the last four rows of table 3.

On the transmit side, the copy latency and total system time drop noticeably, apparently due to a warmer data cache. On the receive side, zero-copy costs ttcp a higher user time than single-copy due to a colder data cache. This is also confirmed by its higher cache miss rate than the case without touching the data. On both sides, zero-copy is still a performance win overall.

6.6. Loaded System Test

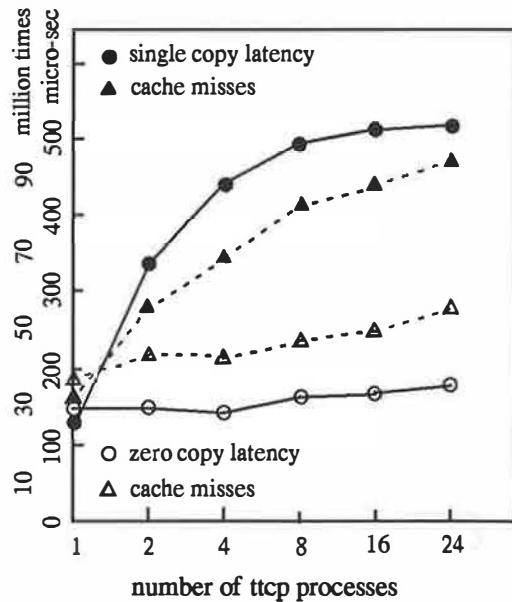
It is imperative to study how zero-copy performs under load compared to single-copy. This is especially

important for servers where there are often multiple tasks running concurrently. On one hand, increasing VM lock contention may hurt zero-copy performance. On the other hand, thread migration caused by higher CPU contention, plus excessive copy operations may greatly reduce the effectiveness of the hardware cache, and affect single-copy performance.

We ran multiple copies of ttcp simultaneously on SPARCserver/1000E to simulate a loaded system. Zero-copy shows little sign of VM lock contentions, and turns out to scale well with the number of running processes. On the receive side, single-copy's performance doesn't change much, possibly due to a cold cache that can't get any worse. Nevertheless, on the transmit side the warmer data cache that helps single-copy performance with only one ttcp running becomes a liability when multiple ttcp processes contend and migrate among different CPUs. This is especially damaging on SPARCserver/1000E, where an access to a remote E-cache is slower than main memory.

The following figure demonstrates this phenomenon. When two copies of ttcp are run, copy time of 8K bytes more than doubles to 325 μ s from 115 μ s. This is also reflected by the number of cache misses. With more copies of ttcp running, single copy latencies increase steadily until saturated around 500 μ s, whereas both the cache misses and latency curves for zero-copy stay flat.

Figure 3: Loaded System Test on SS1000E



7. Conclusions and Future Work

This paper presents an efficient zero-copy implementation for network I/O. It discussed trade-offs among several zero-copy schemes, and contends that our design based on virtual memory page remapping and copy-on-write techniques require the least amount of changes to the existing system and applications, while still offer significant performance gain.

Performance improvements vary somewhat, depending on how efficiently a host's memory cache system can copy data, versus how much overhead MMU operations require. Furthermore, zero-copy works best with other copyless I/O paths, such as memory mapped file, where data is moved to or from disks without CPU copy.

On the transmit side, COW faults are expensive, so is setting up a COW protection on a user buffer, and tearing it down later. Applications need to use a chain of buffers of some sufficiently large size in order to avoid COW faults. Without end-to-end acknowledgments, they don't know when the transport is finished with a buffer. Therefore, COW protections are necessary to ensure data integrity. Nevertheless, if zero-copy is combined with asynchronous I/O facility, so that applications get notified when a buffer is released by the transport and can be safely reused, COW protection won't be necessary. The saving is significant as demonstrated by the following test.

Table 6. Transmit side zero-copy w/o copy-on-write protections

	Processing time (μs)	Latency (μs)	CPU sys (sec)
SS20-UP-0-COW	255	115	46
SS20-UP-w/o COW	190	72	34
SS20-UP-1	360	220	65
SS1000E-0-COW	235	140	42
SS1000E-w/o COW	135	65	24
SS1000E-1	265	115	47.5

With network adaptors capable of calculating TCP checksum, normally networking software no longer needs to have access to user data. This presents another opportunity for further performance improvement. Physical data pages may simply flow through the kernel domain without being mapped. On the transmit side, it saves all the kernel mapping cache work. On the receive side, unmapped and unnamed pages may simply be mapped into user address space and named appropriately. The amount of page remap-

ping and renaming work will be only half of what is required now. The potential benefit is even greater for NFS***, as physical pages carrying client data can be renamed and written out to disk without ever being mapped.

Acknowledgments

Neal Nuckolls first proposed the project. Erik Nordmark offered many great insights and suggestions. Bruce Curtis implemented the hardware checksum support and gave valuable critics. Denny Gentry and Jerry Chen provided the ATM hardware and device driver support.

I'd also like to thank Bill Shannon and the VM group for reviewing the VM code, Mike Tracy for scrutinizing the STREAMS code, Wolfgang Thaler for proof reading the paper, and Helen Vanderberg for correcting my writing. Special thanks to Anil Shivalingiah for introducing me into the VM system many years ago.

Most of all, I thank my dear wife Wendy for sparing my many, many weekends at work.

References

- [1] M. Accetta, R. Baron, D. Golub, R. Rashid, A. Tevanian, and M. Young. "Mach: A New Kernel Foundation for UNIX Development," *Proceedings of the Summer 1986 USENIX Technical Conference and Exhibition*, June 1986.
- [2] D. R. Cheriton. "The V distributed system," *Communications of the ACM*, vol.31, no.3, March 1988.
- [3] D. D. Clark, V. Jacobson, J. Romkey, and H. Salwen. "An Analysis of TCP Processing Overhead," *IEEE Communications Magazine*, 23-29, June 1989.
- [4] E. Cooper, P. Steenkiste, R. Sansom, and B. Zill. "Protocol Implementation on the Nectar Communication Processor," *Proceedings of SIGCOMM '90 Conference on Comm. Architectures, Protocols and Applications*, Aug. 1994.
- [5] C. Dalton, G. Watson, D. Banks, C. Calamvokis, A. Edwards, and J. Lumley. "Afterburner - A network-independent card provides architectural support for high-performance protocols," *IEEE Network*, July 1993.

***. Due to a buffer alignment issue described in Section 3.2.4, zero-copy is currently not supported in NFS.

- [6] P. Druschel and L Peterson. "Fbufs: A High-Bandwidth Cross-Domain Transfer Facility," *Proceedings of the Fourteenth ACM Symposium on Operating Systems Principles*, Dec. 1993.
- [7] P. Druschel, L. Peterson, and B. Davie. "Experiences with a High-Speed Network Adaptor: A Software Perspective," *Proceedings of SIGCOMM'94 Conference on Comm. Architectures, Protocols and Applications*, Aug. 1994.
- [8] R. Fitzgerald and R. F. Rashid. "The Integration of Virtual Memory Management and Interprocess Communication in Accent," *ACM Transactions on Computer Systems*, 4.2, May 1986.
- [9] R. Gingell, J. Moran and W. Shannon. "Virtual Memory Architecture in SunOS," *Proceedings of the USENIX Conference*, May 1987.
- [10] J. Kay and J. Pasquale. "Measurement, Analysis, and Improvement of UDP/IP Throughput for the DECstation 5000," *Proceedings of the Winter USENIX Conference*, Jan. 1993.
- [11] K. Kleinpaste, P. Steenkiste, and B. Zill. "Software Support for Outboard Buffering and Checksumming," *Proceedings of SIGCOMM'95 Conference on Comm. Architectures, Protocols and Applications*, Aug. 1995.
- [12] J. Moran. "SunOS Virtual Memory Implementation", *Proceedings of EUUG Conference*. London England, Spring 1988.
- [13] J. Smith and C. Brendan S. Traw. "Giving Applications Access to Gb/s Networking," *IEEE Network*, July 1993.
- [14] P. Steenkiste. "A Systematic Approach to Host Interface Design for High-Speed Networks," *IEEE Computer*, March 1994.
- [15] B. Traw. "Applying Architectural Parallelism to High Performance Network Subsystems," *Ph.D. Dissertation, University of Pennsylvania*, 1995.
- [16] S.-Y. Tzou and D. P. Anderson. "The Performance of Message-passing using Restricted Virtual Memory Remapping," *Software - Practice and Experience* vol. 21, March 1991.
- [17] A. Wolman, G. Voelker, and C. Thekkath. "Latency Analysis of TCP on an ATM Network," *Proceedings of the Winter USENIX Conference*, Jan. 1994.

Author Information

H.K. Jerry Chu is a Staff Engineer with the Internet Engineering Group at Sun Microsystems. He has been with Sun since 1987, working in many different areas of the operating system kernel, including file systems, VM, low-level porting and system bring-up. His latest venture is in the networking world.

Jerry received his B.S. in Mathematics from National Taiwan University in 1979. While in graduate school, he gave up his youthful dream of becoming a mathematician, and found consolation in computer programming. He received an M.S. in Statistics and an M.S. in Computer Science from Stanford University in 1982 and 1984 respectively. He can be reached electronically at *jerry.chu@Eng.Sun.COM*.

A Performance Comparison of UNIX Operating Systems on the Pentium

Kevin Lai and Mary Baker

Stanford University

Abstract

This paper evaluates the performance of three popular versions of the UNIX operating system on the x86 architecture: Linux, FreeBSD, and Solaris. We evaluate the systems using freely available micro- and application benchmarks to characterize the behavior of their operating system services. We evaluate the currently available major releases of the systems "as-is," without any performance tuning.

Our results show that the x86 operating systems and system libraries we tested fail to deliver the Pentium's full memory write performance to applications. On small-file workloads, Linux is an order of magnitude faster than the other systems. On networking software, FreeBSD provides two to three times higher bandwidth than Linux. In general, Solaris performance usually lies between that of the other two systems.

Although each operating system out-performs the others in some area, we conclude that no one system offers clearly better overall performance. Other factors, such as extra features, ease of installation, or freely available source code, are more convincing reasons for choosing a particular system.

1. Introduction

Many research, development, and product groups that have traditionally run on a UNIX workstation-based computing platform are now moving to a PC-based platform. Organizations can afford to purchase many more PCs than workstations on their equipment budgets. The x86 architecture's low cost, good performance, and expandability give it economies of scale that will reinforce, and be reinforced by, its popularity for at least a few more years.

As part of this transition, these groups must decide whether to move to a PC operating system such as Microsoft Windows, or whether to continue running a UNIX-compatible operating system. For many groups, including our own, dependence on the performance, features, and tools available in the UNIX environment make it sensible to run an x86 implementation of UNIX.

The next step is to choose between the available UNIX-compatible operating systems. In particular,

our group is interested in free implementations of UNIX, because new ideas can be implemented without a non-disclosure agreement and the results can be freely distributed. We were concerned, though, by comments describing the free implementations as toy systems, unsupported and with poor performance and reliability. This argument has been used against Linux especially, since its source is not derived from as respected an ancestor as BSD UNIX 4.4. We decided to compare a few of the systems ourselves to determine the validity of these comments. This paper presents our results.

We benchmarked Linux, FreeBSD, and Solaris. Linux and FreeBSD are the most popular free implementations of UNIX that run on our hardware. Solaris is the least expensive commercial implementation known to support the hardware on our system. (Several other popular systems, such as NetBSD and BSDI's BSD/OS, do not currently support our SCSI controller.) We evaluated only the most recent major releases of the systems currently available, since these are the most relevant and accessible versions for most people.

Our benchmarks are by no means exhaustive; we only measure the performance of tasks and workloads that are important to us. The benchmarks test system call latency; context switch latency for varying numbers of processes; memory bandwidth; file system performance; network bandwidth for pipes, UDP and TCP; and NFS performance on a file system workload.

Our results show that:

- Linux has the best performance on file metadata operations because it updates metadata asynchronously;
- FreeBSD has the best network performance;
- Solaris' performance generally lies between that of the other two systems; and
- All three systems' library routines for setting and copying memory fail to deliver the full underlying Pentium memory bandwidth.

Given these mixed performance results, we believe overall performance is not a sufficient argument for choosing one of these operating systems over the others. Performance on specific tasks may make the dif-

ference for some users, but the systems are competitive overall, and particular performance problems are likely to improve in future releases of all three systems.

Other factors may be more important, including extra features, licensing arrangements, ease of installation, and available support. Solaris provides more sophisticated features, including multiprocessor support, than the current free versions of UNIX, and this will be sufficient argument in its favor for many users. The freely available source code and free licensing of Linux and FreeBSD motivate others to choose one of these systems. Ease of installation and the level of available support are also important, and we include a section in this paper on our relative experiences installing and configuring the systems on our hardware. A large user community and free access to software and other resources over the Internet combine to provide reasonable support for the free implementations.

We hope these results will contribute helpful information for those choosing a UNIX-compatible operating system for the PC. We also hope the results, where negative, will reveal areas for improvement in future versions of these systems.

The remaining sections of this paper describe our benchmarking platform and methodology, our results in more detail, and our experiences installing and using the three systems.

2. Benchmarking Platform

Our goal was to compare UNIX operating systems on identical PC hardware performing some standard tasks of interest to us. The relative performance of the systems on identical tasks is more important to us than the absolute best performance that could be achieved for any individual system through system-specific tuning. For comparison purposes, and because we only have source code available for two of the three systems, our benchmarking methodology is the "black box" approach. We usually attempt to explain curious results through external testing and benchmarking rather than investigations of kernel code or profiling.

2.1 Operating Systems

For operating systems, we chose UNIX systems that have a reasonably large user base and development group, run on our hardware, and cost less than \$100 in the summer of 1995. Linux, FreeBSD and Solaris met these criteria.

Linux is a free version of UNIX distributed under the terms of the GNU General Public License. Roughly speaking, this means that works derived from the Linux kernel must be distributed with source code and a fee can only be charged for the transfer of a copy and not ownership of a copy or licensing of a copy. Linux was created by Linus Torvalds when he was a student at the University of Helsinki in Finland. Since then, many other developers have contributed to it. Its code is not derived from BSD or System V, but has features of both and is also generally compliant with Posix.1.

FreeBSD is a free version of UNIX distributed under the terms of a University of California license. This license requires that the copyright notice be included in both source and binary forms of any distribution and any advertising must mention that the product contains University of California, Berkeley code. FreeBSD is derived from the BSD 4.4-lite release by the Computer Systems Research Group at U.C. Berkeley. Like Linux, many developers have contributed to it. It is fully compatible with BSD-style API programs.

Solaris is a commercial version of UNIX developed by Sun Microsystems, Inc. As a commercial system, it costs money, but we purchased it on CD-ROM for \$99; this made it cheaper than other available commercial versions of UNIX. Including program development utilities, the total cost was \$244. No source code is included. Solaris is mainly a System-V-based UNIX, but includes BSD-compatibility header files and libraries. Solaris runs on both the x86 and Sparc architectures. It has a fully preemptive multi-threaded kernel and support for multi-processor systems.

Of these systems, we chose the most recent major release that was commonly available at our cut-off date of October 31, 1995. Consequently, we did not test unreleased or beta versions. For Linux, we used version 1.2.8 from the Slackware Distribution. For FreeBSD, we used version 2.0.5R. For Solaris, we used version 2.4. Of course, subsequent versions of any of these systems may perform very differently from the versions we tested.

2.2 Hardware

We chose the most cost-effective high performance hardware that was available to us in May, 1995. Our benchmarking platform is `tnt.stanford.edu`, an Intel Pentium P54C-100MHz system with 32 megabytes of main memory and two 2-gigabyte disks. It has a standard 10-Megabit/second Ethernet card (3Com Etherlink III 3c509). The motherboard is an Intel Plato. The disk controller is an NCR 53c810 PCI

SCSI card, which has no on-board cache. One disk is a Quantum Empire 2100 SCSI disk, and the other is an HP 3725 SCSI disk.

On the first disk we installed the various operating systems, each in its own partition. The partitioning of the disk is shown in Table 1. We made each partition 200 megabytes more than the minimum recommended for each OS. Since we installed Linux last, its partition received the remainder of the disk and is larger than it needs to be. (Otherwise, its partition would be the same size as FreeBSD's.)

OS	Version	Size (megabytes)
DOS/Windows	6.2/3.1	250
Solaris	2.4	700
FreeBSD	2.0.5R	400
Linux	1.2.8	600

TABLE 1. Disk Partitioning: This table lists the versions of the operating systems benchmarked and shows how tnt.stanford.edu's disk is partitioned.

We used the second disk to ensure that the unequal partitioning of the first disk does not affect our file system performance results. All benchmarks that manipulate files refer to files on this second disk. We create a fresh 200-megabyte file system on this second disk between different benchmarks, but use the same file system for different iterations of the same benchmark. In this way, each of the systems has the benefit of a fresh file system for its use, but any problems it suffers from its management of that file system during the benchmark will remain.

3. Benchmark Overview

We took our benchmarks from a variety of sources. The system call, context switch, and file create/delete microbenchmarks are derived from those John Ousterhout used in [Ousterhout 90] to compare the effect of RISC and CISC architectures on operating system performance. The Modified Andrew Benchmark, developed from the Andrew Benchmark written by M. Satyanarayanan at CMU [Howard 88], was also used in Ousterhout's experiments. To get a more complete picture of context switch and memory system performance, we rewrote Ousterhout's benchmarks in those areas, and we modified the Modified Andrew Benchmark for better portability.

To test file system bandwidth and seek performance, we used Tim Bray's *bonnie* benchmark [Bray 90].

Our network benchmarks are a combination of some of the network benchmarks from Larry McVoy's

lmbench package [McVoy 95] and the *tftp* TCP/IP benchmarking program.

We tied all of these benchmarks together with Tcl scripts [Tcl 90] and ran each benchmark program twenty times. Most of the benchmark programs themselves also loop several times over their respective routines, and we report the average result for the total number of iterations. All benchmarks were executed in single-user mode. When run in multi-user mode, the benchmarks exhibited slightly higher variance.

4. System Call

We measure system call performance since the system call is one of the basic mechanisms by which the operating system provides functionality to applications. Our results show that Linux has the fastest basic system call, followed by FreeBSD and then Solaris.

We estimate system call time by calling *getpid()* in a loop. We then divide the total time by the number of calls. This is an optimistic estimate of the time to make a system call, because the loop allows successive *getpid()* calls to benefit from data and instructions cached the first time through the loop. Furthermore, *getpid()* does so little work in the kernel that all of the application data and code can remain in the processor cache. Given the few instructions executed in the loop and the small amount of data accessed, the entire loop could execute in an eight-kilobyte instruction and eight-kilobyte data processor cache. This estimate, although optimistic, is fine for our purposes, because we want to measure the relative performance of the systems on the same hardware.

OS	Time (microseconds)	Std Dev	Norm.
Linux	2.31	0.10%	1.00
FreeBSD	2.62	0.08%	0.88
Solaris	3.52	2.95%	0.66

TABLE 2. System Call: This table lists the time to make the *getpid()* system call, averaged over twenty runs of 100,000 iterations each. Lower times are better. The Norm. column lists the speed of the benchmark, normalized to the best time among the systems. This gives a proportional ranking for the systems in which higher numbers are better.

Table 2 shows the results. Examination of the source code for performing a system call reveals that Linux has slightly more optimized assembly instructions than FreeBSD. Solaris' extra features and multi-threaded fully-preemptive kernel contribute to its longer system call time [McVoy 95].

5. Context Switching

Context switch time is important for file and database servers and is increasingly important for Internet servers that must sometimes service hundreds of simultaneous connections. We determined that Linux has the best context switch time of the three systems with fewer than 20 processes, while FreeBSD is faster with more processes. Solaris context switches more slowly in all cases.

We used our own context switching benchmark, `ctx`, based on ideas from the original Ousterhout context switching benchmark, `cswitch`, and Larry McVoy's `lmbench` suite. `ctx` estimates the context switch time by measuring the time to `write()` a byte to another process and then `read()` the one-byte reply. For more than one process, the byte is passed around in a round-robin fashion through a ring of processes. The overhead of the pipe operations is included in our results.

As with the `getpid` benchmark, most, if not all of the code and data in the loop could be cached in the first-level CPU cache, since the Pentium architecture has a physically addressed first- and second-level cache [Anderson 93] that does not need to be flushed during context switch. In addition, the context switch benchmark is written as one program that forks into the required number of processes. Code-sharing

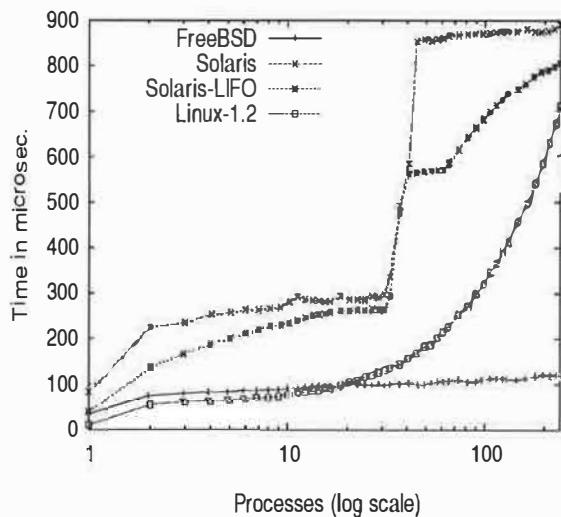


FIGURE 1. Context Switch: This figure shows the time in microseconds to make a context switch as a function of the number of active processes in the system. The results include the overhead of the pipe operations used in the benchmark. Solaris-LIFO passes a byte back and forth through a chain of processes. The other benchmarks pass the byte around a ring of processes. Data points are the average values over twenty runs with 50,000 context switches each. At two processes, the standard deviations were 3% for Linux, 4% for FreeBSD, and 9% for Solaris.

between the processes increases the probability that the entire loop fits into the cache. As with the system call benchmark, this lower-bound estimate is fine for our purposes, since we want to compare the systems.

Figure 1 shows that FreeBSD context switches at almost the same speed no matter how many active processes there are. In contrast, Linux context switching time increases linearly with the number of active processes, suggesting that the Linux scheduler must search an $O(\text{number of processes})$ data structure during a context switch. Aside from the linear time required to traverse this data structure, Linux has very little overhead, so it context switches faster than FreeBSD for fewer than 20 active processes.

Solaris context switches more slowly in all cases. This is in part due to slower pipe performance (as described in Section 9.1). We measured the overhead of sending a byte from a process, through a pipe, and back to the same process. This took 80 microseconds. The time for Solaris to context switch between two active processes is 220 microseconds. Therefore, without the pipe overhead, the estimated time to context switch would be about 140 microseconds. For the same number of processes, FreeBSD and Linux context switch in 80 and 55 microseconds, respectively. The additional overhead is largely due to the extra work that Solaris' multi-threaded fully preemptive kernel scheduler must perform [McVoy 95].

Another interesting result for Solaris is the large increase in context switch time at about 32 processes. We hypothesized that a system resource overflows at that point. In order to test this, we changed the `ctx` benchmark so that its processes pass the token in LIFO order, back and forth through a chain of processes. We expected that this would take advantage of a system table with a limited number of elements and show a gradual increase in context switch time per process for more than 32 processes, instead of a steep one. As shown in Figure 1, this is only true for more than 64 processes. We still see a sharp increase at 32 processes. This behavior does not occur for Solaris running on other architectures [Bonwick 95], so it is not caused by the machine-independent portion of the Solaris scheduler.

6. Memory Bandwidth

As CPUs become faster without a matching speedup in memory, the time to access memory may dominate the execution time of non-I/O-bound programs, including the operating system. We therefore wanted to know which of these systems best exposes the underlying Pentium memory performance. To do this, we compared the performance of the systems' `libc`

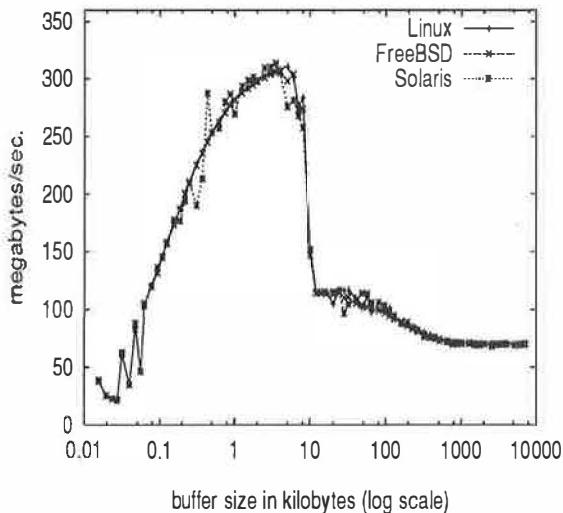


FIGURE 2. Custom Read: This figure shows memory read bandwidth as a function of the size of the buffer read. The humps at 8 kilobytes and 256 kilobytes reveal cache effects. The results are averaged over twenty runs.

`memcpy()` and `memset()` routines. Essentially the same routines are also used by the operating systems themselves. We also wrote our own easily-modifiable custom routines for reading/writing/copying data to help us better understand the behavior of the system library routines.

For all the benchmarks, one or two buffers of varying sizes are used to read, write, or copy data. The same buffers are used over and over again until eight megabytes of data have been transferred, since this gives us direct information about the effects of the hardware caches.

Our results show that none of the systems adequately delivers the Pentium's memory write performance. For example, the Pentium can copy data at over 160 megabytes/second using a prefetching copy routine, yet none of the systems we tested have implemented such a routine. As described below, the prefetching routines address the fact that the Pentium does not have a write-allocate cache. Without this optimization, the same routines copy data at about 40 megabytes/second.

6.1 Memory Read

As shown in Figure 2, the Pentium can read at a peak bandwidth of slightly over 300 megabytes/second from its first-level cache, i.e., it is reading approximately one word every 13ns or four words every 50ns. Since our Pentium runs at 100Mhz, 50ns corresponds to five clock ticks. Given the Pentium architecture's dual issue pipeline, this is a reasonable result.

For buffer sizes larger than 8 kilobytes, the Pentium's performance drops off significantly because that is the size of its first-level data cache.

The next plateau is from approximately 10 kilobytes to 256 kilobytes, where the bandwidth is 110 megabytes/second. This is due to the second-level cache. Finally, read performance levels out at approximately 75 megabytes/second.

6.2 Memory Write

Given the good read performance of the systems, we were initially surprised by the poor `memset()` write bandwidth, which did not reach even 50 megabytes/second (Figure 3).

This poor write performance is due to the lack of a write-allocate cache on the Pentium [Intel 94]. In a write-allocate cache, when a write is done to a line that is not in the cache, that line is brought into the cache while the write is being done, so that later writes to the same line will hit in the cache. We speculated that prefetching the cache lines in software could improve performance on a chip without a write-allocate cache.

In order to test this hypothesis, we coded two versions of a custom memory writing routine, one to do a normal copy and the other to prefetch cache lines as the write is taking place. The results of our non-prefetching custom write benchmark are shown in Figure 4 and are very similar to the system `memset()` results. In comparison, the prefetching version improved the Pentium's performance dramatically, as shown in Figure 5. The peak write bandwidth improved to 310 megabytes/second.

6.3 Memory Copy

As with the `memset()` function, the `memcpy()` routine on the x86 systems has not been optimized to prefetch, so the results for `memcpy()` in Figure 6 resemble those for a custom copy routine without prefetching (Figure 7).

As with the custom write routine, we re-coded the custom copy routine to do prefetching and achieved a peak of over 160 megabytes/second in copy bandwidth, as shown in Figure 8. This is equivalent to 320 megabytes/second in total bandwidth, which approaches the peak set by the custom read routine.

6.4 Memory Anomalies

The spikes at the low end of the figures for all of the custom memory benchmarks are a consequence of the way the memory benchmarks are written. The mem-

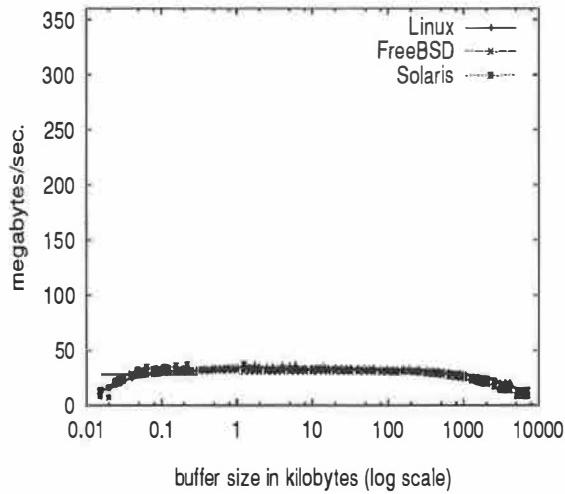


FIGURE 3. Memset: This figure shows memory write bandwidth using `memset()` as a function of buffer size. The results are averaged over twenty runs.

ory benchmark's inner loop actually consists of two loops. One loop performs the appropriate operation on 16 bytes of data per iteration and iterates

$$\left\lfloor \frac{\text{totalBytes}}{16} \right\rfloor$$

times. The other loop performs the same operation to the remaining 0-15 bytes at one iteration per byte. When the buffer size is such that 15 bytes have to be processed in the second loop, the memory bandwidth dips, since the second loop is so much more inefficient than the first.

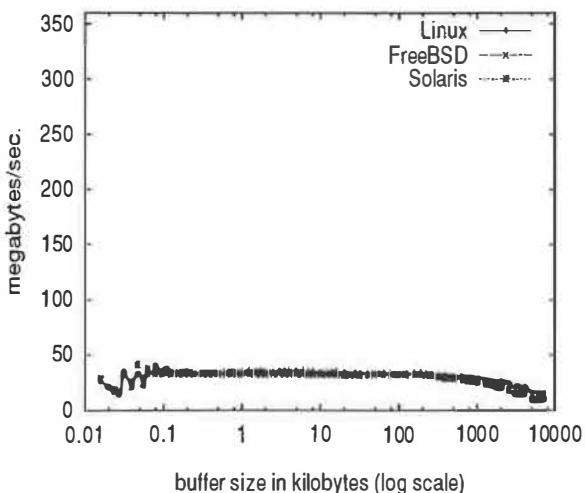


FIGURE 4. Naive Custom Write: This figure shows memory write bandwidth using a custom memory writing routine as a function of buffer size. The results are averaged over twenty runs.

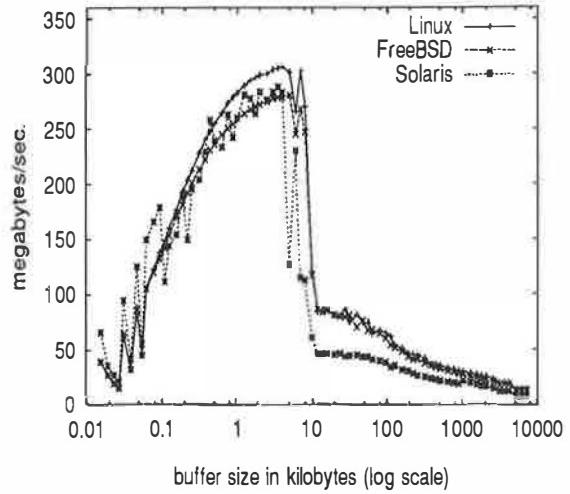


FIGURE 5. Prefetching Custom Write: This figure shows memory write bandwidth with prefetching as a function of buffer size. The results are averaged over twenty runs.

6.5 Summary

Applications programmers rely on the system to shield them from the unnecessary details of the machine while delivering its performance. In this duty, the x86 operating system libraries that we tested fall short in exposing the full memory write bandwidth of the Pentium.

Adding prefetch to memory routines in software used across all the processors in the x86 family is not necessarily appropriate, since some members of the x86 family have a write-allocate cache. Therefore, statically-linking applications with prefetching memory routines might cause these applications to perform worse on some CPUs. However, adding prefetching memory routines to dynamically-linked libraries would allow maximum performance on each machine, because the decision about which library to link with is made at run time. Similarly, adding prefetching memory routines to the kernel allows maximum performance, since the kernel can be compiled separately for individual machines.

7. File System Performance

We benchmarked the ability of the operating systems to satisfy the needs of two types of I/O-intensive workloads. One workload, which includes applications such as video playback and editing and large databases, accesses large files and therefore needs high raw bandwidth and fast seeking. The other workload includes program compilation and accessing, creating and deleting many small files. It therefore stresses the file system's ability to manipulate the file

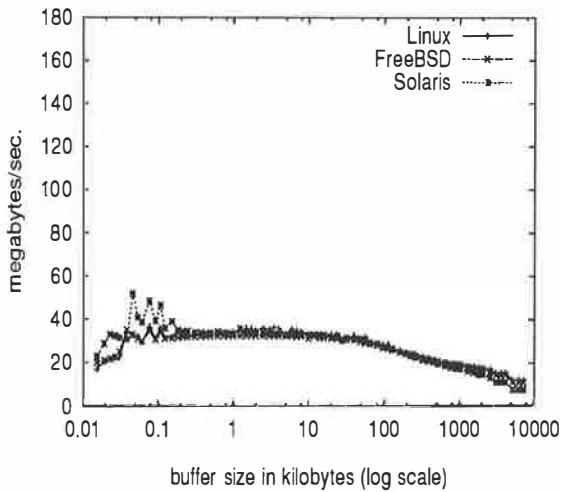


FIGURE 6. *Memcpy*: This figure shows memory copy bandwidth using *memcpy()* as a function of buffer size. The results are averaged over twenty runs.

In order to isolate the differences between operating systems, we used two disks to do the file system benchmarking. The Quantum 2100S contains the operating systems themselves and the code for the benchmarks. We used the HP 3725 as the actual benchmarking disk. We used the same partition for each system and benchmark. After each benchmark (*bonnie*, *crtdel*, MAB), we re-made the file system on that partition to ensure that the previous benchmark could not affect the allocation of blocks during the current benchmark.

All of the systems have a dynamically sized buffer cache that trades physical pages for buffer cache

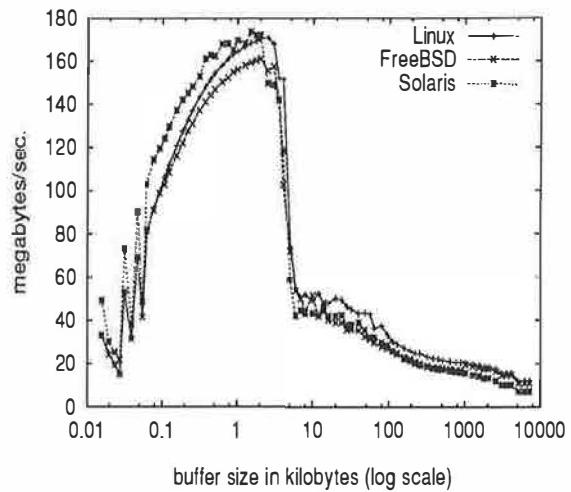


FIGURE 8. *Prefetching Custom Copy*: This figure shows memory copy bandwidth with prefetching as a function of buffer size. The results are averaged over twenty runs.

pages during intensive disk accesses; as a result, they generally do well when the data set accessed is small enough to fit in main memory. Our results show that FreeBSD and Solaris perform well for large files. For small file workloads, characterized by a high percentage of metadata operations, Linux is an order of magnitude faster than the other systems, because it performs file metadata updates asynchronously.

7.1 Large-file Benchmarks

We wanted to test three aspects of large file performance: 1) sequential read bandwidth, 2) sequential write bandwidth, and 3) time to seek to a random block in a file and perform an I/O operation on it. To do this, we used the *bonnie* benchmark, written by Tim Bray. *Bonnie* creates and writes to a file of the user-specified size, reads from it sequentially, and then seeks randomly within it. We ran *bonnie* with file sizes from two to 100 megabytes to test performance for files that do and do not fit in the buffer cache. Unlike some of the other benchmarks we used, *bonnie* performs each of its operations only once per invocation. We invoke it 20 times per file size.

As shown in Figure 9, all three systems cache the file for sizes up to 20 megabytes out of 32 megabytes total on the machine. This is because all three systems allow a trade-off between memory pages and the file cache, so the file cache can grow to accommodate large files.

For files in the buffer cache, FreeBSD reads between 5% and 15% faster than both Linux and Solaris. For files outside of the buffer cache, Solaris has the best read bandwidth. Large sequential

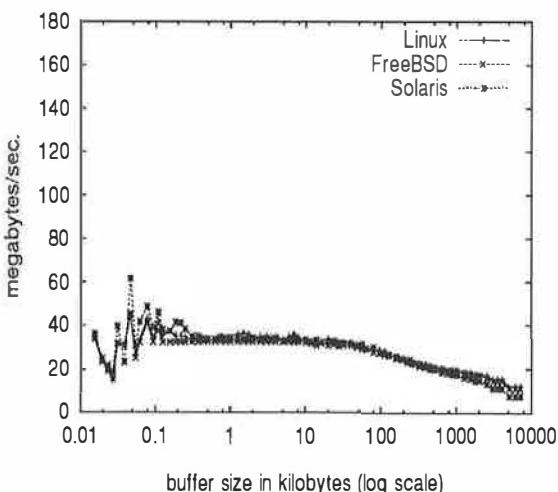


FIGURE 7. *Naive Custom Copy*: This figure shows memory copy bandwidth using a custom copy routine as a function of buffer size. The results are averaged over twenty runs.

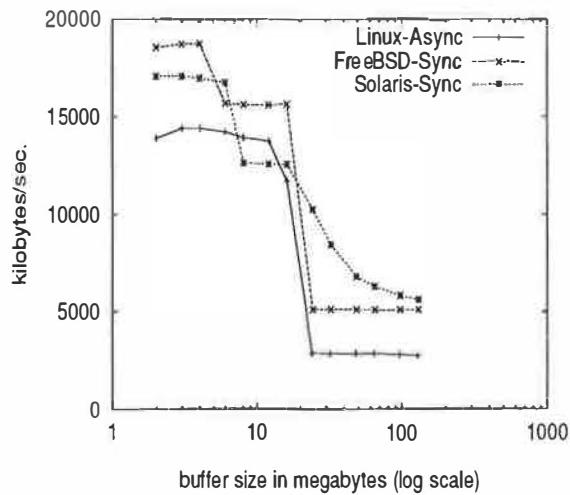


FIGURE 9. **Bonnie Read:** This figure shows file system sequential read bandwidth in Megabytes/second as a function of file size. The results are averaged over twenty runs.

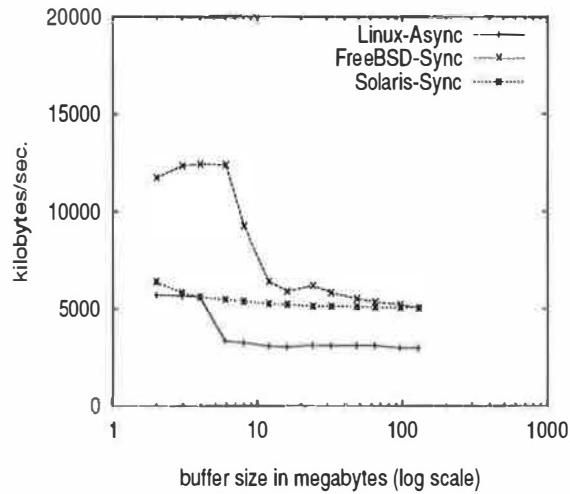


FIGURE 10. **Bonnie Write:** This figure shows file system sequential write bandwidth as a function of file size. The results are averaged over twenty runs.

accesses negate the benefits of an LRU file cache, and Solaris compensates for this better than the other systems. Linux has the worst read bandwidth for files larger than the buffer cache.

The effects of FreeBSD's efficient file cache and Linux's poor large file performance are also apparent in Figure 10. FreeBSD writes files of size less than eight megabytes 50% faster than Solaris or Linux. Linux maintains less than half the write bandwidth of FreeBSD or Solaris for almost all file sizes.

In contrast, Linux and Solaris can perform approximately 50% more random seeks and I/O operations per second than FreeBSD for files inside the file

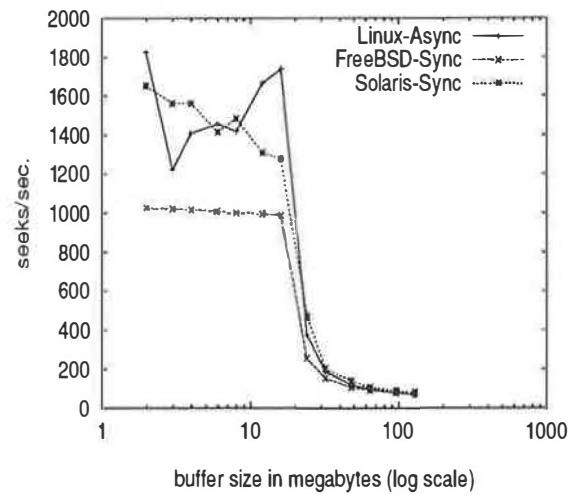


FIGURE 11. **Bonnie Seek:** This figure shows the number of random seeks per second as a function of file size. The results are averaged over twenty runs.

cache, as shown in Figure 11. The *bonnie seek* benchmark seeks to a random block in a file, reads the 8-kilobyte block and then writes it out. All three systems converge to 14ms for random seeks to blocks on disk.

7.2 Small-file and Metadata Benchmarks

To benchmark the ability of these operating systems to deal with many small files, we used *crtdel* from the Ousterhout microbenchmarks. *Crtdel* opens a file, writes some data to it, closes it, opens it again, reads data from it, and deletes it. It mimics the use of a temporary file by a compiler. It stresses the updating of file system metadata such as the inode, directory block and directory inode. We ran it using various file sizes to get a view of metadata overhead versus file data overhead (Figure 12).

Given that we measured the average non-cached seek time of these systems to be 14ms (Figure 11), Linux clearly is not accessing the disk during this benchmark. This is because the Linux file system, ext2fs, uses an asynchronous metadata update policy, unlike the FreeBSD and Solaris file systems. While this gives Linux a performance advantage, it could result in losing more data after a system crash. Some of the synchronous updates in the BSD-derived file systems are intended to help preserve file system consistency in the event of such failures.

FreeBSD does worse on this benchmark than can be explained by its use of synchronous metadata writes. Since both the FreeBSD and Solaris 2.4 file systems are derived from the BSD FFS [McKusick 84], they both use synchronous metadata writes. However,

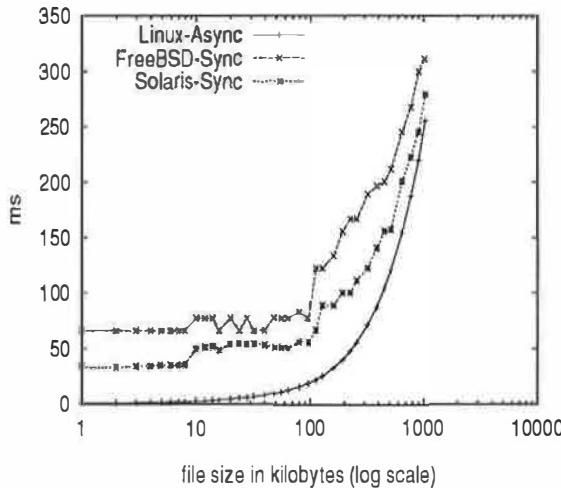


FIGURE 12. File Create/Delete: This figure shows the number of milliseconds to create and delete files as a function of file size. The results are averaged over twenty runs.

Solaris executes `crtdel` in only 34ms (Figure 12), compared to FreeBSD's almost 66ms. The magnitude of FreeBSD's overhead compared to Solaris suggests that it accesses the disk more than is necessary or seeks further. Furthermore, as the amount of data written increases from one kilobyte to one megabyte, the difference between the Solaris `crtdel` time and the FreeBSD `crtdel` time remains almost constant at about 32ms.

We also tested the FreeBSD file system using its optional asynchronous update policy. However, this option appears not to be implemented yet in version 2.5R, since our results for the synchronous and asynchronous modes were identical within the range of experimental error.

8. Modified Andrew Benchmark (MAB)

So far, we have reported the results of microbenchmarks. Microbenchmarks measure particular aspects of a system, but they may not reflect overall system performance under any realistic workload.

As a step towards comparing the operating systems under a typical software engineering workload, we used the Modified Andrew Benchmark (MAB). It consists of five parts: directory creation, file copying, directory stats, file reading, and compilation. To achieve portability and to eliminate the differences in the compilation speed of different compilers, the original MAB includes the source for an early version of `gcc`. This early version of `gcc` is used to compile for the SPUR architecture during the compilation phase. The code generated during the benchmark is never

executed, so the choice of architecture does not matter.

We found we had to make further modifications to MAB, so our results are no longer directly comparable to previously-reported MAB results. The problem with the original MAB is that its version of `ranlib` relies on the system's `ar`, and binary file formats have changed enough that this scheme no longer works. Furthermore, the version of `gcc` in the original MAB is not portable to Linux or System V OSs (such as Solaris). To maintain the spirit of the original MAB, we modified MAB to use a recent version of `gcc` and included a compatible version of GNU's `binutils`, which includes portable versions of `ar`, `ld` and `ranlib`. We configured `gcc` and the `binutils` to generate code for the `x86` architecture under Linux since the `SPUR` architecture is no longer supported as a compilation target.

In this section we report MAB results for a local file system. We report MAB results for accessing remote file systems over NFS in Section 10.

8.1 Local File System

The results of running MAB on a local disk are summarized in Table 3. Linux's first place finish is not surprising, given its performance on the file and disk micro-benchmarks. Linux's asynchronous file metadata updates and its good read performance for the small files (<1 megabyte) used in MAB indicate that it should do well on MAB.

OS	Time (seconds)	Std Dev	Norm.
Linux	43.12	4.10%	1.00
FreeBSD	47.45	1.02%	0.91
Solaris 2.4	54.31	1.93%	0.80

TABLE 3. MAB Local: This table shows the total time to execute MAB on the local file system as averaged over twenty runs.

What is more surprising is FreeBSD's good performance on MAB, given its poor performance in manipulating file meta-data and in reading small files. FreeBSD is competitive with Solaris in each of the benchmark phases, except stating directories, where it exceeds even Linux's performance. FreeBSD keeps a separate attribute cache for the directory information, which is filled in the first phase (directory creation) and is thus accessed in the third phase (directory stats). Linux does not have such a separate attribute cache, so its attribute information is knocked

out of the file data cache during the second (file copying) phase.

9. Network Benchmarks

Faster network technology such as 100 Megabit/second Ethernet is becoming more affordable, while CPUs are becoming memory speed bound. As a result, the limiting factor for network performance is the efficiency of the network protocol implementation. We discovered that none of the x86 systems can fully utilize a 100 Megabit/second Ethernet link, with Linux being two to three times slower than FreeBSD and Solaris.

In most of our network benchmarks we used the loopback interface rather than an actual Ethernet interface. Although this ignores the effect of collisions and other real world effects, we wanted to measure the best possible performance in order to predict these operating systems' performance on a future network.

To isolate possible contributors and detriments to network performance, we tested network performance using three protocols: pipes, UDP, and TCP.

9.1 Pipes

Although pipes are not a network protocol, they require much of the same functionality as a network protocol, such as system calls, context switches and data copying. We measured pipe bandwidth as an upper bound on what network protocols could achieve if there were no other overhead. The pipe benchmark, `bw_pipe`, comes from Larry McVoy's `lmbench` benchmark package. It forks off a child and transfers 50 megabytes in 64-kilobyte chunks between itself and the child.

OS	Bandwidth (megabits/ second)	Std Dev	Norm.
Linux	119.36	1.60%	1.00
FreeBSD	98.03	2.79%	0.82
Solaris 2.4	65.38	1.56%	0.55

TABLE 4. Pipe Bandwidth: This table shows the bandwidth of a pipe as averaged over twenty runs.

From Table 4, we see that Linux and FreeBSD could theoretically keep up with a 100-Megabit/second Ethernet, if the TCP/IP protocols added no additional overhead. Solaris, however, could not keep up. Solaris' slower system calls and context switches do not explain this poor performance. The extra overhead

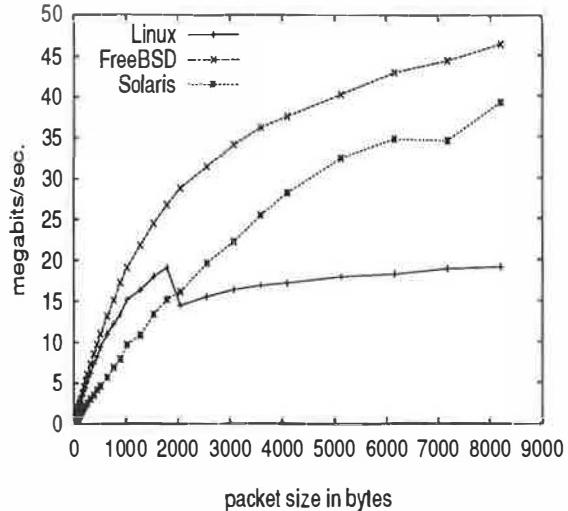


FIGURE 13. UDP: This figure shows UDP bandwidth as a function of packet size when averaged over twenty benchmark runs.

for Solaris pipes is largely due to their implementation on top of System V streams [Kottapurath 95].

9.2 UDP

The UDP protocol is a slightly higher-level protocol than pipes, in that UDP forms packets but does not use time-outs, sequence numbers, and retransmission as in TCP. In order to test UDP bandwidth, we ran `tcp` using a variety of packet sizes, transferring 4 megabytes every iteration. When run as the sender, `tcp` reads data from `stdin`, breaks it up into packets, and sends the packets to the receiver. When run as the receiver, `tcp` reads packets and writes the data to `stdout`. We redirected the output to `/dev/null`.

From Figure 13, we see that FreeBSD achieves a bandwidth of almost 50 megabits/second, meaning that its UDP runs at only 50% of the bandwidth of pipes. Solaris is worse. It achieves a peak bandwidth of 32 megabits/second; just as with FreeBSD, this is 50% of the pipe bandwidth. Linux has the most surprising result. Although it has the best pipe bandwidth, it has the worst UDP performance. Its UDP performance of 16 Megabits/second is only 14% of its pipe bandwidth. Its UDP implementation has a high amount of overhead due to unnecessary copies and inefficient buffer allocation.

9.3 TCP

TCP is one of the most widely used protocols today, forming the basis for many reliable protocols, such as ftp. In order to benchmark TCP, we use `bw_tcp`, which comes from Larry McVoy's `lmbench` bench-

mark package. `Bw_tcp` transfers 3 megabytes from

OS	Bandwidth (megabits/ second)	Std Dev	Norm.
FreeBSD	65.95	2.36%	1.00
Solaris 2.4	60.11	16.34%	0.91
Linux	25.03	5.45%	0.38

TABLE 5. TCP Bandwidth: This table shows the bandwidth of a TCP connection.

one process to another during each iteration using a 48K buffer.

As shown in Table 5, Solaris's TCP performance is not hindered by its poor UDP performance. On the other hand, Linux's TCP implementation is just as slow as its UDP implementation. Our investigations indicate that version 1.2.8 of Linux has a TCP window of only one packet. This severely limits its TCP bandwidth, as our results show.

10. MAB across NFS

To measure network file system performance for the three systems, we ran MAB over NFS, using the three systems as clients. We ran these tests using a Linux 1.2.8 file server and a SunOS 4.1.4 file server. We did not test FreeBSD or Solaris as servers, since we do not have the extra equipment available.

OS	Time (seconds)	Std Dev	Norm.
FreeBSD	53.24	0.87%	1.00
Linux	57.73	2.20%	0.92
Solaris 2.4	58.38	1.36%	0.91

TABLE 6. MAB NFS with Linux Server: This table shows the total time to execute MAB across NFS to a Linux server.

Using a Linux server, the FreeBSD client was the top performer due to its good networking performance. Linux comes in second place with Solaris coming in third.

Overall, the benchmark ran more slowly when accessing the SunOS server rather than the Linux server. The SunOS file server uses a synchronous update policy, as required by the NFS specifications. The Linux file server continues using its asynchronous update policy, and we hypothesize that this explains the difference in performance.

With the SunOS file server, we see somewhat different relative results between the three clients.

FreeBSD's good networking performance again serves it well when connected to a SunOS NFS server. Solaris performs relatively poorly when using the SunOS server instead of the Linux server. Linux's networking code is apparently tuned to work with other Linux hosts and performs miserably when connected to other types of servers.

OS	Time (seconds)	Std Dev	Norm.
FreeBSD	67.60	1.41%	1.00
Solaris 2.4	87.94	3.17%	0.77
Linux	115.06	1.54%	0.59

TABLE 7. MAB NFS w/SunOS Server: This table shows the total time to execute MAB across NFS to a SunOS Server.

11. Other Comments

In testing the performance of these systems, we encountered other differences that may be of interest to those choosing which system to run. Although some of these differences may disappear in later releases, some are a consequence of the policies of the system developers or vendors and therefore may not change in future releases. These differences include installation difficulties, porting differences, and system bugs found while running the benchmarks. All of these areas help indicate the level of support one can expect when using the systems.

In general, the availability of free system source code combined with a large user community seems to have a positive effect on these problems. If the user community contributes drivers and other system software, then that system will work sooner on a wider range of hardware than is possible for a system with only a few developers. Even if no one has contributed a desired feature yet, we can implement and even distribute it ourselves at a minimum of cost. Almost by definition, systems research requires new hardware and software that has not yet made it to the commercial sector. Additionally, a large user community with access to source code will provide support for a system outside of a vendor's or a developer's support, increasing the probability that bugs will be found and fixed quickly and questions answered.

Our installation experiences with the three systems were very different, with Linux being the easiest and Solaris being the most difficult.

Some of the good installation features:

- Installation across the Internet (Linux, FreeBSD)
- WWW installation documentation (Linux, FreeBSD)

Among the problems we encountered:

- Didn't support the (very common) Panasonic/Creative Labs CD-ROM drive (FreeBSD, Solaris)
- Crashed during installation due to a driver incompatibility (FreeBSD, Solaris)
- Obliterated existing boot loader and disk partitions (Solaris)
- Inaccessible or missing system administration documentation (Solaris)

Our experiences porting the benchmarks to the three systems were somewhat more pleasant, with Linux again being the easiest system and Solaris the most difficult. In general, Solaris was the most difficult because there is no Internet repository of Solaris binaries, and there isn't yet a large enough Solaris x86 user community to provide the level of support found for the other systems.

Some of the good porting features:

- BSD and System V compatibility (Linux)
- Automatic installation of commonly used free software like `gcc`, `emacs`, and `tcsh` (Linux, FreeBSD)
- Internet repository of pre-compiled binaries (Linux, FreeBSD)

Some of the porting difficulties:

- No installed compiler (Solaris)
- Only an old and buggy pre-compiled `gcc` available on the Internet (Solaris)

All of the systems had problems running the benchmarks, with the most irritating problem being that the Linux 1.2.8 NFS server requires that clients connect on a privileged port. FreeBSD 2.0.5 clients do not do this by default.

12. Conclusions

No one system dominates our benchmarks. Linux does well on system calls, context switching, and pipe bandwidth. Its performance on small-file workloads with intensive metadata manipulation is an order of magnitude faster than the other systems. Linux also does well when communicating with a Linux NFS server. However, Linux has poor overall networking performance and poor NFS performance when connected to a SunOS NFS server.

FreeBSD has better networking and NFS performance than the other systems. It performs well on large files but not on small files. It does well on the Modified Andrew Benchmark both remotely and locally.

Solaris has poor system call, context switch and pipe performance. It reads large files efficiently but does poorly when the Modified Andrew Benchmark is run locally.

An inherent disadvantage of our "black box" benchmarking approach is that it cannot conclusively explain all of the performance differences in these systems. In many cases, it merely exposes the differences. In addition, using microbenchmarks isolates the areas of both good and bad performance, but microbenchmarks cannot predict overall application performance. Despite the differences on the microbenchmarks, the systems' overall performance on the MAB workload is much closer.

13. Future Work

Benchmarking operating systems that are under active development is always a work in progress. As we write this paper, new versions of all of these systems are about to be released with several changes in their performance. The latest development version of the Linux kernel (1.3.40) is a good example. It has very fast context switching (10 microseconds for two active processes with very little slowdown as the number of active processes increases). Its NFS performance has also improved. The next release version of FreeBSD (2.1) will offer ordered asynchronous metadata updates to improve small-file performance while helping maintain file system consistency during a crash. The next version of Solaris (2.5) will have faster context switching and better performance in general.

Architectural support for counting operating system events such as TLB misses [Chen 95] can reveal more about the workings of an operating system than using timers alone. We plan to apply some of those techniques to the systems that interest us.

14. Benchmark Source Code Availability

Our benchmarking package is available at <http://plastique.stanford.edu/bench.html>.

15. Acknowledgments

We thank Larry McVoy for his many helpful comments. We thank Jeff Bonwick, Sherif Kottapurath, Dean Long, and Behfar Razavi for their answers to questions about Solaris. We thank Stuart Cheshire,

Elliot Poger, Mendel Rosenblum, Jonathan Stone, Diane Tang, and especially Darrell Long for their comments on the paper. We thank the authors of all the benchmarks we used. This work was supported by a grant from the Reid and Polly Anderson Faculty Scholar Fund at Stanford University.

16. References

- [Anderson 93] Don Anderson and Tom Shanley, *Pentium Processor System Architecture*, MindShare Press, 1993.
- [Anderson 91] Thomas Anderson, Henry Levy, Brian Bershad, and Edward Lazowska, "The Interaction of Architecture and Operating System Design." *ASPLOS-IV*, April 1991.
- [Bonwick 95] Jeff Bonwick, personal communication, November 1995.
- [Bray 90] Tim Bray, Bonnie source code, 1990.
- [Chen 93] J. Bradley Chen and Brian N. Bershad, "The Impact of Operating System Structure on Memory System Performance," *Proceedings of the Fourteenth International Symposium on Operating Systems Principles*, pp. 120-133, December 1993.
- [Chen 95] J. Bradley Chen, Yasuhiro Endo, Kee Chan, David Mazieres, Antonio Dias, Margo Seltzer, and Michael D. Smith, "The Measured Performance of Personal Computer Operating System." To appear in the *Proceedings of the Fifteenth International Symposium on Operating Systems Principles*, December 1995.
- [FreeBSD 95] Various Authors, FreeBSD Home Page, <http://www.freebsd.org/>, 1995.
- [Howard 88] J. Howard, et al. "Scale and Performance in a Distributed File System." *ACM Transactions on Computer Systems*, Vol. 6, No. 1, February 1988, pp. 51-81.
- [Intel 94] Intel Corporation, "The Pentium Family User's Manual, Volume 3: Architecture and Programming Manual." Intel Literature Sales, P.O. Box 7641, Mt. Prospect, IL 60056-7641, 1994.
- [Kottapurath 95] Sherif Kottapurath, personal communication, November 1995.
- [LDP 95] Various Authors, Linux Documentation Project, <http://sunsite.unc.edu/mdw/welcome.html>, 1995.
- [McKusick 84] Marshall K. McKusick, "A Fast File System for Unix," *ACM Transactions on Computer Systems* 2(3) pp. 181-197, 1984.

[McVoy 95] Larry McVoy and Carl Staelin, "lmbench: Portable tools for performance analysis," To appear in *Proceedings for the 1996 Usenix Technical Conference*, January 1996.

[Ousterhout 90] John K. Ousterhout, "Why Aren't Operating Systems Getting Faster As Fast As Hardware?" *Proceedings of the 1990 Summer Usenix Conference*, pp. 247-256, June 1990.

[Rashid 88] Richard F. Rashid, et al., "Machine-Independent Virtual Memory Management for Paged Uniprocessor and Multiprocessor Architectures," *IEEE Transactions on Computers*, Vol. 37 No. 8, pp. 896-908, August 1988.

[Tcl 90] John K. Ousterhout, *Tcl and the Tk Toolkit*, Addison-Wesley, Reading, Massachusetts, 1994.

[Welsh 94] Matt Welsh, *The Linux Bible*, Yggdrasil Computing Incorporated, 1994

17. Biographical Information

Kevin Lai is a Master's student at Stanford University. He received his B.A. in Computer Science in 1992 from U.C. Berkeley. His interests include performance measurement, operating systems, and operating system support for mobile computing.

Mary Baker is an assistant professor in the Departments of Computer Science and Electrical Engineering at Stanford University. Her interests include operating systems, distributed systems, and software fault tolerance. She received her Ph.D. in computer science in 1994 from U.C. Berkeley.

The authors' email addresses are
{laik, mgbaker}@cs.stanford.edu.

lmbench: Portable tools for performance analysis

Larry McVoy

Silicon Graphics, Inc.

Carl Staelin

Hewlett-Packard Laboratories

Abstract

lmbench is a micro-benchmark suite designed to focus attention on the basic building blocks of many common system applications, such as databases, simulations, software development, and networking. In almost all cases, the individual tests are the result of analysis and isolation of a customer's actual performance problem. These tools can be, and currently are, used to compare different system implementations from different vendors. In several cases, the benchmarks have uncovered previously unknown bugs and design flaws. The results have shown a strong correlation between memory system performance and overall performance. lmbench includes an extensible database of results from systems current as of late 1995.

1. Introduction

lmbench provides a suite of benchmarks that attempt to measure the most commonly found performance bottlenecks in a wide range of system applications. These bottlenecks have been identified, isolated, and reproduced in a set of small micro-benchmarks, which measure system latency and bandwidth of data movement among the processor and memory, network, file system, and disk. The intent is to produce numbers that real applications can reproduce, rather than the frequently quoted and somewhat less reproducible marketing performance numbers.

The benchmarks focus on latency and bandwidth because performance issues are usually caused by latency problems, bandwidth problems, or some combination of the two. Each benchmark exists because it captures some unique performance problem present in one or more important applications. For example, the TCP latency benchmark is an accurate predictor of the Oracle distributed lock manager's performance, the memory latency benchmark gives a strong indication of Verilog simulation performance, and the file system latency benchmark models a critical path in software development.

lmbench was developed to identify and evaluate system performance bottlenecks present in many machines in 1993-1995. It is entirely possible that computer architectures will have changed and advanced enough in the next few years to render parts

of this benchmark suite obsolete or irrelevant.

lmbench is already in widespread use at many sites by both end users and system designers. In some cases, lmbench has provided the data necessary to discover and correct critical performance problems that might have gone unnoticed. lmbench uncovered a problem in Sun's memory management software that made all pages map to the same location in the cache, effectively turning a 512 kilobyte (K) cache into a 4K cache.

lmbench measures only a system's ability to transfer data between processor, cache, memory, network, and disk. It does not measure other parts of the system, such as the graphics subsystem, nor is it a MIPS, MFLOPS, throughput, saturation, stress, graphics, or multiprocessor test suite. It is frequently run on multiprocessor (MP) systems to compare their performance against uniprocessor systems, but it does not take advantage of any multiprocessor features.

The benchmarks are written using standard, portable system interfaces and facilities commonly used by applications, so lmbench is portable and comparable over a wide set of Unix systems. lmbench has been run on AIX, BSDI, HP-UX, IRIX, Linux, FreeBSD, NetBSD, OSF/1, Solaris, and SunOS. Part of the suite has been run on Windows/NT as well.

lmbench is freely distributed under the Free Software Foundation's General Public License [Stallman89], with the additional restriction that results may be reported only if the benchmarks are unmodified.

2. Prior work

Benchmarking and performance analysis is not a new endeavor. There are too many other benchmark suites to list all of them here. We compare lmbench to a set of similar benchmarks.

- **I/O (disk) benchmarks:** IOstone [Park90] wants to be an I/O benchmark, but actually measures the memory subsystem; all of the tests fit easily in the cache. IObench [Wolman89] is a systematic file system and disk benchmark, but it is complicated and unwieldy. In [McVoy91] we reviewed many I/O benchmarks and found them all lacking because they took too long to run and were too complex a solution to a fairly simple

problem. We wrote a small, simple I/O benchmark, `lmbench` that measures sequential and random I/O far faster than either IOStone or IOBench. As part of [McVoy91] the results from `lmbench` were checked against IOBench (as well as some other Sun internal I/O benchmarks). `lmbench` proved to be more accurate than any of the other benchmarks. At least one disk vendor routinely uses `lmbench` to do performance testing of its disk drives.

Chen and Patterson [Chen93, Chen94] measure I/O performance under a variety of workloads that are automatically varied to test the range of the system's performance. Our efforts differ in that we are more interested in the CPU overhead of a single request, rather than the capacity of the system as a whole.

- **Berkeley Software Distribution's microbench suite:** The BSD effort generated an extensive set of test benchmarks to do regression testing (both quality and performance) of the BSD releases. We did not use this as a basis for our work (although we used ideas) for the following reasons: (a) missing tests — such as memory latency, (b) too many tests, the results tended to be obscured under a mountain of numbers, and (c) wrong copyright — we wanted the Free Software Foundation's General Public License.
- **Ousterhout's Operating System benchmark:** [Ousterhout90] proposes several system benchmarks to measure system call latency, context switch time, and file system performance. We used the same ideas as a basis for our work, while trying to go farther. We measured a more complete set of primitives, including some hardware measurements; went into greater depth on some of the tests, such as context switching; and went to great lengths to make the benchmark portable and extensible.
- **Networking benchmarks:** Netperf measures networking bandwidth and latency and was written by Rick Jones of Hewlett-Packard. `lmbench` includes a smaller, less complex benchmark that produces similar results.

`tccp` is a widely used benchmark in the Internet community. Our version of the same benchmark routinely delivers bandwidth numbers that are within 2% of the numbers quoted by `tccp`.

- **McCalpin's stream benchmark:** [McCalpin95] has memory bandwidth measurements and results for a large number of high-end systems. We did not use these because we discovered them only after we had results using our versions. We will probably include McCalpin's benchmarks in `lmbench` in the future.

In summary, we rolled our own because we wanted simple, portable benchmarks that accurately measured a wide variety of operations that we consider crucial to performance on today's systems. While portions of other benchmark suites include similar work, none includes all of it, few are as portable,

and almost all are far more complex. Less filling, tastes great.

3. Benchmarking notes

3.1. Sizing the benchmarks

The proper sizing of various benchmark parameters is crucial to ensure that the benchmark is measuring the right component of system performance. For example, memory-to-memory copy speeds are dramatically affected by the location of the data: if the size parameter is too small so the data is in a cache, then the performance may be as much as ten times faster than if the data is in memory. On the other hand, if the memory size parameter is too big so the data is paged to disk, then performance may be slowed to such an extent that the benchmark seems to 'never finish.'

`lmbench` takes the following approach to the cache and memory size issues:

- All of the benchmarks that could be affected by cache size are run in a loop, with increasing sizes (typically powers of two) until some maximum size is reached. The results may then be plotted to see where the benchmark no longer fits in the cache.
- The benchmark verifies that there is sufficient memory to run all of the benchmarks in main memory. A small test program allocates as much memory as it can, clears the memory, and then strides through that memory a page at a time, timing each reference. If any reference takes more than a few microseconds, the page is no longer in memory. The test program starts small and works forward until either enough memory is seen as present or the memory limit is reached.

3.2. Compile time issues

The GNU C compiler, `gcc`, is the compiler we chose because it gave the most reproducible results across platforms. When `gcc` was not present, we used the vendor-supplied `cc`. All of the benchmarks were compiled with optimization `-O` except the benchmarks that calculate clock speed and the context switch times, which must be compiled without optimization in order to produce correct results. No other optimization flags were enabled because we wanted results that would be commonly seen by application writers.

All of the benchmarks were linked using the default manner of the target system. For most if not all systems, the binaries were linked using shared libraries.

3.3. Multiprocessor issues

All of the multiprocessor systems ran the benchmarks in the same way as the uniprocessor systems. Some systems allow users to pin processes to a particular CPU, which sometimes results in better cache reuse. We do not pin processes because it defeats the

MP scheduler. In certain cases, this decision yields interesting results discussed later.

3.4. Timing issues

- Clock resolution:** The benchmarks measure the elapsed time by reading the system clock via the `gettimeofday` interface. On some systems this interface has a resolution of 10 milliseconds, a long time relative to many of the benchmarks which have results measured in tens to hundreds of microseconds. To compensate for the coarse clock resolution, the benchmarks are hand-tuned to measure many operations within a single time interval lasting for many clock ticks. Typically, this is done by executing the operation in a small loop, sometimes unrolled if the operation is exceedingly fast, and then dividing the loop time by the loop count.

- Caching:** If the benchmark expects the data to be in the cache, the benchmark is typically run several times; only the last result is recorded.

If the benchmark does not want to measure cache performance it sets the size parameter larger than the cache. For example, the `bcopy` benchmark by default copies 8 megabytes to 8 megabytes, which largely defeats any second-level cache in use today. (Note that the benchmarks are not trying to defeat the file or process page cache, only the hardware caches.)

- Variability:** The results of some benchmarks, most notably the context switch benchmark, had a tendency to vary quite a bit, up to 30%. We suspect that the operating system is not using the same set of physical pages each time a process is created and we are seeing the effects of collisions in the external caches. We compensate by running the benchmark in a loop and taking the minimum result. Users interested in the most accurate data are advised to verify the results on their own platforms.

Many of the results included in the database were donated by users and were not created by the authors. Good benchmarking practice suggests that one should

run the benchmarks as the only user of a machine, without other resource intensive or unpredictable processes or daemons.

3.5. Using the lmbench database

`lmbench` includes a database of results that is useful for comparison purposes. It is quite easy to build the source, run the benchmark, and produce a table of results that includes the run. All of the tables in this paper were produced from the database included in `lmbench`. This paper is also included with `lmbench` and may be reproduced incorporating new results. For more information, consult the file `lmbench-HOWTO` in the `lmbench` distribution.

4. Systems tested

`lmbench` has been run on a wide variety of platforms. This paper includes results from a representative subset of machines and operating systems. Comparisons between similar hardware running different operating systems can be very illuminating, and we have included a few examples in our results.

The systems are briefly characterized in Table 1. Please note that the list prices are very approximate as is the year of introduction. The SPECInt92 numbers are a little suspect since some vendors have been “optimizing” for certain parts of SPEC. We try and quote the original SPECInt92 numbers where we can.

4.1. Reading the result tables

Throughout the rest of this paper, we present tables of results for many of the benchmarks. All of the tables are sorted, from best to worst. Some tables have multiple columns of results and those tables are sorted on only one of the columns. The sorted column's heading will be in **bold**.

5. Bandwidth benchmarks

By bandwidth, we mean the rate at which a particular facility can move data. We attempt to measure the data movement ability of a number of different

Name used	Vendor & model	Multi or Uni	Operating System	CPU	Mhz	Year	SPEC Int92	List price
IBM PowerPC	IBM 43P	Uni	AIX 3.?	MPC604	133	'95	176	15k
IBM Power2	IBM 990	Uni	AIX 4.?	Power2	71	'93	126	110k
FreeBSD/i386	ASUS P55TP4XE	Uni	FreeBSD 2.1	Pentium	133	'95	190	3k
HP K210	HP 9000/859	MP	HP-UX B.10.01	PA 7200	120	'95	167	35k
SGI Challenge	SGI Challenge	MP	IRIX 6.2- α	R4400	200	'94	140	80k
SGI Indigo2	SGI Indigo2	Uni	IRIX 5.3	R4400	200	'94	135	15k
Linux/Alpha	DEC Cabriolet	Uni	Linux 1.3.38	Alpha 21064A	275	'94	189	9k
Linux/i386	Triton/EDO RAM	Uni	Linux 1.3.28	Pentium	120	'95	155	5k
Linux/i686	Intel Alder	Uni	Linux 1.3.37	Pentium Pro	200	'95	320	7k
DEC Alpha@150	DEC 3000/500	Uni	OSF1 3.0	Alpha 21064	150	'93	84	35k
DEC Alpha@300	DEC 8400 5/300	MP	OSF1 3.2	Alpha 21164	300	'95	341	? 250k
Sun Ultra	Sun Ultra	Uni	SunOS 5.5	UltraSPARC	167	'95	250	21k
Sun SC1000	Sun SC1000	MP	SunOS 5.5- β	SuperSPARC	50	'92	65	35k
Solaris/i686	Intel Alder	Uni	SunOS 5.5.1	Pentium Pro	133	'95	215	5k
Unixware/i686	Intel Aurora	Uni	Unixware 5.4.2	Pentium Pro	200	'95	320	7k

Table 1. System descriptions.

facilities: library `bcopy`, hand-unrolled `bcopy`, direct-memory read and write (no copying), pipes, TCP sockets, the `read` interface, and the `mmap` interface.

5.1. Memory bandwidth

Data movement is fundamental to any operating system. In the past, performance was frequently measured in MFLOPS because floating point units were slow enough that microprocessor systems were rarely limited by memory bandwidth. Today, floating point units are usually much faster than memory bandwidth, so many current MFLOP ratings can not be maintained using memory-resident data; they are “cache only” ratings.

We measure the ability to copy, read, and write data over a varying set of sizes. There are too many results to report all of them here, so we concentrate on large memory transfers.

We measure copy bandwidth two ways. The first is the user-level library `bcopy` interface. The second is a hand-unrolled loop that loads and stores aligned 8-byte words. In both cases, we took care to ensure that the source and destination locations would not map to the same lines if any of the caches were direct-mapped. In order to test memory bandwidth rather than cache bandwidth, both benchmarks copy an 8M¹ area to another 8M area. (As secondary caches reach 16M, these benchmarks will have to be resized to reduce caching effects.)

The copy results actually represent one-half to one-third of the memory bandwidth used to obtain those results since we are reading and writing memory. If the cache line size is larger than the word stored, then the written cache line will typically be read before it is written. The actual amount of memory bandwidth used varies because some architectures have special instructions specifically designed for the `bcopy` function. Those architectures will move twice as much memory as reported by this benchmark; less advanced architectures move three times as much memory: the memory read, the memory read because it is about to be overwritten, and the memory written.

The `bcopy` results reported in Table 2 may be correlated with John McCalpin’s `stream` [McCalpin95] benchmark results in the following manner: the `stream` benchmark reports all of the memory moved whereas the `bcopy` benchmark reports the bytes copied. So our numbers should be approximately one-half to one-third of his numbers.

Memory reading is measured by an unrolled loop that sums up a series of integers. On most (perhaps all) systems measured the integer size is 4 bytes. The loop is unrolled such that most compilers generate code that uses a constant offset with the load, resulting

in a load and an add for each word of memory. The add is an integer add that completes in one cycle on all of the processors. Given that today’s processor typically cycles at 10 or fewer nanoseconds (ns) and that memory is typically 200-1,000 ns per cache line, the results reported here should be dominated by the memory subsystem, not the processor add unit.

The memory contents are added up because almost all C compilers would optimize out the whole loop when optimization was turned on, and would generate far too many instructions without optimization. The solution is to add up the data and pass the result as an unused argument to the “finish timing” function.

Memory reads represent about one-third to one-half of the `bcopy` work, and we expect that pure reads should run at roughly twice the speed of `bcopy`. Exceptions to this rule should be studied, for exceptions indicate a bug in the benchmarks, a problem in `bcopy`, or some unusual hardware.

System	Bcopy unrolled	libc	Memory read	write
IBM Power2	242	171	205	364
Sun Ultra	85	167	129	152
DEC Alpha@300	85	80	120	123
HP K210	78	57	117	126
Unixware/i866	65	58	235	88
Solaris/i866	52	48	159	71
DEC Alpha@150	46	45	79	91
Linux/i686	42	56	208	56
FreeBSD/i586	39	42	73	83
Linux/Alpha	39	39	73	71
Linux/i586	38	42	74	75
SGI Challenge	35	36	65	67
SGI Indigo2	31	32	69	66
IBM PowerPC	21	21	63	26
Sun SC1000	17	15	38	31

Table 2. Memory bandwidth (MB/s)

Memory writing is measured by an unrolled loop that stores a value into an integer (typically a 4 byte integer) and then increments the pointer. The processor cost of each memory operation is approximately the same as the cost in the read case.

The numbers reported in Table 2 are not the raw hardware speed in some cases. The Power2² is capable of up to 800M/sec read rates [McCalpin95] and HP PA RISC (and other prefetching) systems also do better if higher levels of code optimization used and/or the code is hand tuned.

The Sun libc `bcopy` in Table 2 is better because they use a hardware specific `bcopy` routine that uses instructions new in SPARC V9 that were added specifically for memory movement.

The Pentium Pro read rate in Table 2 is much higher than the write rate because, according to Intel,

¹ Some of the PCs had less than 16M of available memory; those machines copied 4M.

² Someone described this machine as a \$1,000 processor on a \$99,000 memory subsystem.

the write transaction turns into a read followed by a write to maintain cache consistency for MP systems.

5.2. IPC bandwidth

Interprocess communication bandwidth is frequently a performance issue. Many Unix applications are composed of several processes communicating through pipes or TCP sockets. Examples include the groff documentation system that prepared this paper, the X Window System, remote file access, and World Wide Web servers.

Unix pipes are an interprocess communication mechanism implemented as a one-way byte stream. Each end of the stream has an associated file descriptor; one is the write descriptor and the other the read descriptor. TCP sockets are similar to pipes except they are bidirectional and can cross machine boundaries.

Pipe bandwidth is measured by creating two processes, a writer and a reader, which transfer 50M of data in 64K transfers. The transfer size was chosen so that the overhead of system calls and context switching would not dominate the benchmark time. The reader prints the timing results, which guarantees that all data has been moved before the timing is finished.

TCP bandwidth is measured similarly, except the data is transferred in 1M page aligned transfers instead of 64K transfers. If the TCP implementation supports it, the send and receive socket buffers are enlarged to 1M, instead of the default 4-60K. We have found that setting the transfer size equal to the socket buffer size produces the greatest throughput over the most implementations.

System	Libc bcopy	pipe	TCP
HP K210	57	93	34
Linux/i686	56	89	18
IBM Power2	171	84	10
Linux/Alpha	39	73	9
Unixware/i686	58	68	-1
Sun Ultra	167	61	51
DEC Alpha@300	80	46	11
Solaris/i686	48	38	20
DEC Alpha@150	45	35	9
SGI Indigo2	32	34	22
Linux/i586	42	34	7
IBM PowerPC	21	30	17
FreeBSD/i586	42	23	13
SGI Challenge	36	17	31
Sun SC1000	15	9	11

Table 3. Pipe and local TCP bandwidth (MB/s)

bcopy is important to this test because the pipe write/read is typically implemented as a bcopy into the kernel from the writer and then a bcopy from the kernel to the reader. Ideally, these results would be approximately one-half of the bcopy results. It is possible for the kernel bcopy to be faster than the C library bcopy since the kernel may have access to

bcopy hardware unavailable to the C library.

It is interesting to compare pipes with TCP because the TCP benchmark is identical to the pipe benchmark except for the transport mechanism. Ideally, the TCP bandwidth would be as good as the pipe bandwidth. It is not widely known that the majority of the TCP cost is in the bcopy, the checksum, and the network interface driver. The checksum and the driver may be safely eliminated in the loopback case and if the costs have been eliminated, then TCP should be just as fast as pipes. From the pipe and TCP results in Table 3, it is easy to see that Solaris and HP-UX have done this optimization.

Bcopy rates in Table 3 can be lower than pipe rates because the pipe transfers are done in 64K buffers, a size that frequently fits in caches, while the bcopy is typically an 8M-to-8M copy, which does not fit in the cache.

In Table 3, the SGI Indigo2, a uniprocessor, does better than the SGI MP on pipe bandwidth because of caching effects - in the UP case, both processes share the cache; on the MP, each process is communicating with a different cache.

All of the TCP results in Table 3 are in loopback mode — that is both ends of the socket are on the same machine. It was impossible to get remote networking results for all the machines included in this paper. We are interested in receiving more results for identical machines with a dedicated network connecting them. The results we have for over the wire TCP bandwidth are shown below.

System	Network	TCP bandwidth
SGI PowerChallenge	hippi	79.3
Sun Ultra	100baseT	9.5
HP 9000/735	fddi	8.8
FreeBSD/i586	100baseT	7.9
SGI Indigo2	10baseT	.9
HP 9000/735	10baseT	.9
Linux/i586@90Mhz	10baseT	.7

Table 4. Remote TCP bandwidth (MB/s)

The SGI using 100MB/s Hippi is by far the fastest in Table 4. The SGI Hippi interface has hardware support for TCP checksums and the IRIX operating system uses virtual memory tricks to avoid copying data as much as possible. For larger transfers, SGI Hippi has reached 92MB/s over TCP.

100baseT is looking quite competitive when compared to FDDI in Table 4, even though FDDI has packets that are almost three times larger. We wonder how long it will be before we see gigabit ethernet interfaces.

5.3. Cached I/O bandwidth

Experience has shown us that reusing data in the file system page cache can be a performance issue. This section measures that operation through two

interfaces, `read` and `mmap`. The benchmark here is not an I/O benchmark in that no disk activity is involved. We wanted to measure the overhead of reusing data, an overhead that is CPU intensive, rather than disk intensive.

The `read` interface copies data from the kernel's file system page cache into the process's buffer, using 64K buffers. The transfer size was chosen to minimize the kernel entry overhead while remaining realistically sized.

The difference between the `bcopy` and the `read` benchmarks is the cost of the file and virtual memory system overhead. In most systems, the `bcopy` speed should be faster than the `read` speed. The exceptions usually have hardware specifically designed for the `bcopy` function and that hardware may be available only to the operating system.

The `read` benchmark is implemented by rereading a file (typically 8M) in 64K buffers. Each buffer is summed as a series of integers in the user process. The summing is done for two reasons: for an apples-to-apples comparison the memory-mapped benchmark needs to touch all the data, and the file system can sometimes transfer data into memory faster than the processor can read the data. For example, SGI's XFS can move data into memory at rates in excess of 500M per second, but it can move data into the cache at only 68M per second. The intent is to measure performance delivered to the application, not DMA performance to memory.

System	Libc <code>bcopy</code>	File <code>read</code>	Memory <code>read</code>	File <code>mmap</code>
IBM Power2	171	187	205	106
HP K210	57	88	117	52
Sun Ultra1	167	85	129	101
DEC Alpha@300	80	67	120	78
Unixware/i686	58	62	235	200
Solaris/i686	48	52	159	94
DEC Alpha@150	45	40	79	50
Linux/i686	56	40	208	36
IBM PowerPC	21	40	63	51
SGI Challenge	36	36	65	56
SGI Indigo2	32	32	69	44
FreeBSD/i386	42	30	73	53
Linux/Alpha	39	24	73	18
Linux/i586	42	23	74	9
Sun SC1000	15	20	38	28

Table 5. File vs. memory bandwidth (MB/s)

The `mmap` interface provides a way to access the kernel's file cache without copying the data. The `mmap` benchmark is implemented by mapping the entire file (typically 8M) into the process's address space. The file is then summed to force the data into the cache.

In Table 5, a good system will have `File read` as fast as (or even faster than) `Libc bcopy` because as the file system overhead goes to zero, the file reread case

is virtually the same as the library `bcopy` case. However, file reread can be faster because the kernel may have access to `bcopy` assist hardware not available to the C library. Ideally, `File mmap` performance should approach `Memory read` performance, but `mmap` is often dramatically worse. Judging by the results, this looks to be a potential area for operating system improvements.

In Table 5 the Power2 does better on file reread than `bcopy` because it takes full advantage of the memory subsystem from inside the kernel. The `mmap` reread is probably slower because of the lower clock rate; the page faults start to show up as a significant cost.

It is surprising that the Sun Ultra1 was able to `bcopy` at the high rates shown in Table 2 but did not show those rates for file reread in Table 5. HP has the opposite problem, they get file reread faster than `bcopy`, perhaps because the kernel `bcopy` has access to hardware support.

The Unixware system has outstanding `mmap` reread rates, better than systems of substantially higher cost. Linux needs to do some work on the `mmap` code.

6. Latency measurements

Latency is an often-overlooked area of performance problems, possibly because resolving latency issues is frequently much harder than resolving bandwidth issues. For example, memory bandwidth may be increased by making wider cache lines and increasing memory "width" and interleave, but memory latency can be improved only by shortening paths or increasing (successful) prefetching. The first step toward improving latency is understanding the current latencies in a system.

The latency measurements included in this suite are memory latency, basic operating system entry cost, signal handling cost, process creation times, context switching, interprocess communication, file system latency, and disk latency.

6.1. Memory read latency background

In this section, we expend considerable effort to define the different memory latencies and to explain and justify our benchmark. The background is a bit tedious but important, since we believe the memory latency measurements to be one of the most thought-provoking and useful measurements in `lmbench`.

The most basic latency measurement is memory latency since most of the other latency measurements can be expressed in terms of memory latency. For example, context switches require saving the current process state and loading the state of the next process. However, memory latency is rarely accurately measured and frequently misunderstood.

Memory read latency has many definitions; the most common, in increasing time order, are memory chip cycle time, processor-pins-to-memory-and-back time, load-in-a-vacuum time, and back-to-back-load time.

- **Memory chip cycle latency:** Memory chips are rated in nanoseconds; typical speeds are around 60ns. A general overview on DRAM architecture may be found in [Hennessy96]. The specific information we describe here is from [Toshiba94] and pertains to the THM361020AS-60 module and TC514400AJS DRAM used in SGI workstations. The 60ns time is the time from $\overline{\text{RAS}}$ assertion to the when the data will be available on the DRAM pins (assuming $\overline{\text{CAS}}$ access time requirements were met). While it is possible to get data out of a DRAM in 60ns, that is not all of the time involved. There is a precharge time that must occur after every access. [Toshiba94] quotes 110ns as the random read or write cycle time and this time is more representative of the cycle time.

- **Pin-to-pin latency:** This number represents the time needed for the memory request to travel from the processor's pins to the memory subsystem and back again. Many vendors have used the pin-to-pin definition of memory latency in their reports. For example, [Fenwick95] while describing the DEC 8400 quotes memory latencies of 265ns; a careful reading of that paper shows that these are pin-to-pin numbers. In spite of the historical precedent in vendor reports, this definition of memory latency is misleading since it ignores actual delays seen when a load instruction is immediately followed by a use of the data being loaded. The number of additional cycles inside the processor can be significant and grows more significant with today's highly pipelined architectures.

It is worth noting that the pin-to-pin numbers include the amount of time it takes to charge the lines going to the SIMMs, a time that increases with the (potential) number of SIMMs in a system. More SIMMs mean more capacitance which requires longer charge times. This is one reason why personal computers frequently have better memory latencies than workstations: the PCs typically have less memory capacity.

- **Load-in-a-vacuum latency:** A load in a vacuum is the time that the processor will wait for one load that must be fetched from main memory (i.e., a cache miss). The "vacuum" means that there is no other activity on the system bus, including no other loads. While this number is frequently used as the memory latency, it is not very useful. It is basically a "not to exceed" number important only for marketing reasons. Some architects point out that since most processors implement nonblocking loads (the load does not cause a stall until the data is used), the perceived load latency may be much less than the real latency. When pressed, however, most will admit that cache misses occur in bursts, resulting in perceived latencies

of at least the load-in-a-vacuum latency.

- **Back-to-back-load latency:** Back-to-back-load latency is the time that each load takes, assuming that the instructions before and after are also cache-missing loads. Back-to-back loads may take longer than loads in a vacuum for the following reason: many systems implement something known as *critical word first*, which means that the subblock of the cache line that contains the word being loaded is delivered to the processor before the entire cache line has been brought into the cache. If another load occurs quickly enough after the processor gets restarted from the current load, the second load may stall because the cache is still busy filling the cache line for the previous load. On some systems, such as the current implementation of UltraSPARC, the difference between back to back and load in a vacuum is about 35%.

lmbench measures back-to-back-load latency because it is the only measurement that may be easily measured from software and because we feel that it is what most software developers consider to be memory latency. Consider the following C code fragment:

```
p = head;
while (p->p_next)
    p = p->p_next;
```

On a DEC Alpha, the loop part turns into three instructions, including the load. A 300 Mhz processor has a 3.33ns cycle time, so the loop could execute in slightly less than 10ns. However, the load itself takes 400ns on a 300 Mhz DEC 8400. In other words, the instructions cost 10ns but the load stalls for 400. Another way to look at it is that 400/3.3, or 121, nondependent, nonloading instructions following the load would be needed to hide the load latency. Because superscalar processors typically execute multiple operations per clock cycle, they need even more useful operations between cache misses to keep the processor from stalling.

This benchmark illuminates the tradeoffs in processor cache design. Architects like large cache lines, up to 64 bytes or so, because the prefetch effect of gathering a whole line increases hit rate given reasonable spatial locality. Small stride sizes have high spatial locality and should have higher performance, but large stride sizes have poor spatial locality causing the system to prefetch useless data. So the benchmark provides the following insight into negative effects of large line prefetch:

- Multi-cycle fill operations are typically atomic events at the caches, and sometimes block other cache accesses until they complete.
- Caches are typically single-ported. Having a large line prefetch of unused data causes extra bandwidth demands at the cache, and can cause increased access latency for normal cache accesses.

In summary, we believe that processors are so fast that the average load latency for cache misses will be closer to the back-to-back-load number than to the load-in-a-vacuum number. We are hopeful that the industry will standardize on this definition of memory latency.

6.2. Memory read latency

The entire memory hierarchy can be measured, including on-board data cache latency and size, external data cache latency and size, and main memory latency. Instruction caches are not measured. TLB miss latency can also be measured, as in [Saavedra92], but we stopped at main memory. Measuring TLB miss time is problematic because different systems map different amounts of memory with their TLB hardware.

The benchmark varies two parameters, array size and array stride. For each size, a list of pointers is created for all of the different strides. Then the list is walked thus:

```
mov r4, (r4) # C code: p = *p;
```

The time to do about 1,000,000 loads (the list wraps) is measured and reported. The time reported is pure latency time and may be zero even though the load instruction does not execute in zero time. Zero is defined as one clock cycle; in other words, the time reported is **only** memory latency time, as it does not include the instruction execution time. It is assumed that all processors can do a load instruction in one processor cycle (not counting stalls). In other words, if the processor cache load time is 60ns on a 20ns processor, the load latency reported would be 40ns, the additional 20ns is for the load instruction itself.³ Processors that can manage to get the load address out to the address pins before the end of the load cycle get some free time in this benchmark (we don't know of any processors that do that).

This benchmark has been validated by logic analyzer measurements on an SGI Indy by Ron Minnich while he was at the Maryland Supercomputer Research Center.

Results from the memory latency benchmark are plotted as a series of data sets as shown in Figure 1. Each data set represents a stride size, with the array size varying from 512 bytes up to 8M or more. The curves contain a series of horizontal plateaus, where each plateau represents a level in the memory hierarchy. The point where each plateau ends and the line rises marks the end of that portion of the memory hierarchy (e.g., external cache). Most machines have similar memory hierarchies: on-board cache, external cache, main memory, and main memory plus TLB miss costs. There are variations: some processors are

³ In retrospect, this was a bad idea because we calculate the clock rate to get the instruction execution time. If the clock rate is off, so is the load time.

DEC alpha@182mhz memory latencies

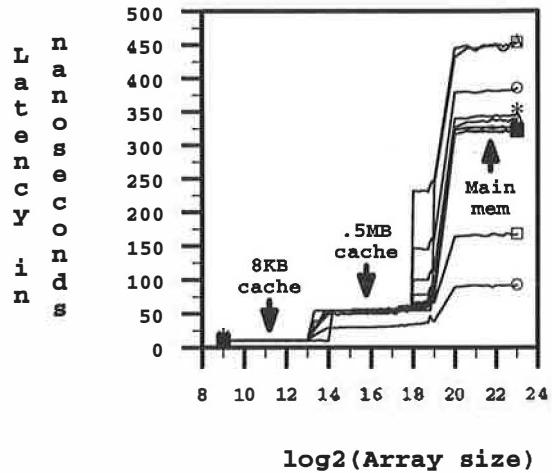


Figure 1. Memory latency

missing a cache, while others add another cache to the hierarchy. For example, the Alpha 8400 has two on-board caches, one 8K and the other 96K.

The cache line size can be derived by comparing curves and noticing which strides are faster than main memory times. The smallest stride that is the same as main memory speed is likely to be the cache line size because the strides that are faster than memory are getting more than one hit per cache line.

Figure 1 shows memory latencies on a nicely made machine, a DEC Alpha. We use this machine as the example because it shows the latencies and sizes of the on-chip level 1 and motherboard level 2 caches, and because it has good all-around numbers, especially considering it can support a 4M level 2 cache. The on-board cache is 2^{13} bytes or 8K, while the external cache is 2^{19} bytes or 512K.

Table 6 shows the cache size, cache latency, and main memory latency as extracted from the memory latency graphs. The graphs and the tools for extracting the data are included with lmbench. It is worthwhile to plot all of the graphs and examine them since the table is missing some details, such as the DEC Alpha 8400 processor's second 96K on-chip cache.

We sorted Table 6 on level 2 cache latency because we think that many applications will fit in the level 2 cache. The HP and IBM systems have only one level of cache so we count that as both level 1 and level 2. Those two systems have remarkable cache performance for caches of that size. In both cases, the cache delivers data in one clock cycle after the load instruction.

HP systems usually focus on large caches as close as possible to the processor. A older HP multiprocessor system, the 9000/890, has a 4M, split I&D, 2 way

System	Level 1 cache			Level 2 cache		Memory latency
	Clk.	lat.	size	lat.	size	
HP K210	8	8	256K	--	--	349
IBM Power2	14	13	256K	--	--	260
Unixware/i686	5	5	8K	25	256K	175
Linux/i686	5	10	8K	30	256K	179
Sun Ultra	6	6	16K	42	512K	270
Linux/Alpha	3	6	8K	46	96K	357
Solaris/i686	7	14	8K	? 48	256K	281
SGI Indigo2	5	8	16K	64	2M	1170
SGI Challenge	5	8	16K	64	4M	1189
DEC Alpha@300	3	3	8K	66	4M	400
DEC Alpha@150	6	12	8K	67	512K	291
FreeBSD/i386	7	7	8K	95	512K	182
Linux/i586	8	8	8K	107	256K	150
Sun SC1000	20	20	8K	140	1M	1236
IBM PowerPC	7	6	16K	164	? 512K	394

Table 6. Cache and memory latency (ns)

set associative cache, accessible in one clock (16ns). That system is primarily a database server.

The IBM focus is on low latency, high bandwidth memory. The IBM memory subsystem is good because all of memory is close to the processor, but has the weakness that it is extremely difficult to evolve the design to a multiprocessor system.

The 586 and PowerPC motherboards have quite poor second level caches, the caches are not substantially better than main memory.

The Pentium Pro and Sun Ultra second level caches are of medium speed at 5-6 clocks latency each. 5-6 clocks seems fast until it is compared against the HP and IBM one cycle latency caches of similar size. Given the tight integration of the Pentium Pro level 2 cache, it is surprising that it has such high latencies.

The 300Mhz DEC Alpha has a rather high 22 clock latency to the second level cache which is probably one of the reasons that they needed a 96K level 1.5 cache. SGI and DEC have used large second level caches to hide their long latency from main memory.

6.3. Operating system entry

Entry into the operating system is required for many system facilities. When calculating the cost of a facility, it is useful to know how expensive it is to perform a nontrivial entry into the operating system.

We measure nontrivial entry into the system by repeatedly writing one word to `/dev/null`, a pseudo device driver that does nothing but discard the data. This particular entry point was chosen because it has never been optimized in any system that we have measured. Other entry points, typically `getpid` and `gettimeofday`, are heavily used, heavily optimized, and sometimes implemented as user-level library routines rather than system calls. A write to the `/dev/null` driver will go through the system

call table to `write`, verify the user area as readable, look up the file descriptor to get the vnode, call the vnode's `write` function, and then return.

System	system call
Linux/Alpha	2
Linux/i586	2
Linux/i686	3
Unixware/i686	4
Sun Ultra	5
FreeBSD/i586	6
Solaris/i686	7
DEC Alpha@300	9
Sun SC1000	9
HP K210	10
SGI Indigo2	11
DEC Alpha@150	11
IBM PowerPC	12
IBM Power2	16
SGI Challenge	24

Table 7. Simple system call time (microseconds)

Linux is the clear winner in the system call time. The reasons are twofold: Linux is a uniprocessor operating system, without any MP overhead, and Linux is a small operating system, without all of the "features" accumulated by the commercial offers.

Unixware and Solaris are doing quite well, given that they are both fairly large, commercially oriented operating systems with a large accumulation of "features."

6.4. Signal handling cost

Signals in Unix are a way to tell another process to handle an event. They are to processes as interrupts are to the CPU.

Signal handling is often critical to layered systems. Some applications, such as databases, software development environments, and threading libraries provide an operating system-like layer on top of the operating system, making signal handling a critical path in many of these applications.

`lmbench` measures both signal installation and signal dispatching in two separate loops, within the context of one process. It measures signal handling by installing a signal handler and then repeatedly sending itself the signal.

Table 8 shows the signal handling costs. Note that there are no context switches in this benchmark; the signal goes to the same process that generated the signal. In real applications, the signals usually go to another process, which implies that the true cost of sending that signal is the signal overhead plus the context switch overhead. We wanted to measure signal and context switch overheads separately since context switch times vary widely among operating systems.

SGI does very well on signal processing, especially since their hardware is of an older generation

System	sigaction	sig handler
SGI Indigo2	4	7
SGI Challenge	4	9
HP K210	4	13
FreeBSD/i386	4	21
Linux/i386	4	22
Unixware/i386	6	25
IBM Power2	10	27
Solaris/i386	9	45
IBM PowerPC	10	52
Linux/i586	7	52
DEC Alpha@300	6	59
Linux/Alpha	13	138

Table 8. Signal times (microseconds)

than many of the others.

The Linux/Alpha signal handling numbers are so poor that we suspect that this is a bug, especially given that the Linux/x86 numbers are quite reasonable.

6.5. Process creation costs

Process benchmarks are used to measure the basic process primitives, such as creating a new process, running a different program, and context switching. Process creation benchmarks are of particular interest in distributed systems since many remote operations include the creation of a remote process to shepherd the remote operation to completion. Context switching is important for the same reasons.

- **Simple process creation.** The Unix process creation primitive is `fork`, which creates a (virtually) exact copy of the calling process. Unlike VMS and some other operating systems, Unix starts any new process with a `fork`. Consequently, `fork` and/or `execve` should be fast and “light,” facts that many have been ignoring for some time.

`lmbench` measures simple process creation by creating a process and immediately exiting the child process. The parent process waits for the child process to exit. The benchmark is intended to measure the overhead for creating a new thread of control, so it includes the `fork` and the `exit` time.

The benchmark also includes a `wait` system call in the parent and context switches from the parent to the child and back again. Given that context switches of this sort are on the order of 20 microseconds and a system call is on the order of 5 microseconds, and that the entire benchmark time is on the order of a millisecond or more, the extra overhead is insignificant. Note that even this relatively simple task is very expensive and is measured in milliseconds while most of the other operations we consider are measured in microseconds.

- **New process creation.** The preceding benchmark did not create a new application; it created a copy of the old application. This benchmark measures the cost of creating a new process and changing that

process into a new application, which forms the basis of every Unix command line interface, or shell. `lmbench` measures this facility by forking a new child and having that child execute a new program — in this case, a tiny program that prints “hello world” and exits.

The startup cost is especially noticeable on (some) systems that have shared libraries. Shared libraries can introduce a substantial (tens of milliseconds) startup cost.

System	fork & exit	fork, exec & exit	fork, exec sh -c & exit
Linux/Alpha	0.7	3	12
Linux/i386	0.4	5	14
Linux/i586	0.9	5	16
Unixware/i386	0.9	5	10
DEC Alpha@300	2.0	6	16
IBM PowerPC	2.9	8	50
SGI Indigo2	3.1	8	19
IBM Power2	1.2	8	16
FreeBSD/i386	2.0	11	19
HP K210	3.1	11	20
DEC Alpha@150	4.6	13	39
SGI Challenge	4.0	14	24
Sun Ultra	3.7	20	37
Solaris/i386	4.5	22	46
Sun SC1000	14.0	69	281

Table 9. Process creation time (milliseconds)

- **Complicated new process creation.** When programs start other programs, they frequently use one of three standard interfaces: `popen`, `system`, and/or `execvp`. The first two interfaces start a new process by invoking the standard command interpreter, `/bin/sh`, to start the process. Starting programs this way guarantees that the shell will look for the requested application in all of the places that the user would look — in other words, the shell uses the user’s `$PATH` variable as a list of places to find the application. `execvp` is a C library routine which also looks for the program using the user’s `$PATH` variable.

Since this is a common way of starting applications, we felt it was useful to show the costs of the generality.

We measure this by starting `/bin/sh` to start the same tiny program we ran in the last case. In Table 9 the cost of asking the shell to go look for the program is quite large, frequently ten times as expensive as just creating a new process, and four times as expensive as explicitly naming the location of the new program.

The results that stand out in Table 9 are the poor Sun Ultra 1 results. Given that the processor is one of the fastest, the problem is likely to be software. There is room for substantial improvement in the Solaris process creation code.

6.6. Context switching

Context switch time is defined here as the time needed to save the state of one process and restore the state of another process.

Context switches are frequently in the critical performance path of distributed applications. For example, the multiprocessor versions of the IRIX operating system use processes to move data through the networking stack. This means that the processing time for each new packet arriving at an idle system includes the time needed to switch in the networking process.

Typical context switch benchmarks measure just the minimal context switch time — the time to switch between two processes that are doing nothing but context switching. We feel that this is misleading because there are frequently more than two active processes, and they usually have a larger working set (cache footprint) than the benchmark processes.

Other benchmarks frequently include the cost of the system calls needed to force the context switches. For example, Ousterhout's context switch benchmark measures context switch time plus a read and a write on a pipe. In many of the systems measured by lmbench, the pipe overhead varies between 30% and 300% of the context switch time, so we were careful to factor out the pipe overhead.

- **Number of processes.** The context switch benchmark is implemented as a ring of two to twenty processes that are connected with Unix pipes. A token is passed from process to process, forcing context switches. The benchmark measures the time needed to pass the token two thousand times from process to process. Each transfer of the token has two costs: the context switch, and the overhead of passing the token. In order to calculate just the context switching time, the benchmark first measures the cost of passing the token through a ring of pipes in a single process. This overhead time is defined as the cost of passing the token and is not included in the reported context switch time.

- **Size of processes.** In order to measure more realistic context switch times, we add an artificial variable size "cache footprint" to the switching processes. The cost of the context switch then includes the cost of restoring user-level state (cache footprint). The cache footprint is implemented by having the process allocate an array of data⁴ and sum the array as a series of integers after receiving the token but before passing the token to the next process. Since most systems will cache data across context switches, the working set for the benchmark is slightly larger than the number of processes times the array size.

It is worthwhile to point out that the overhead mentioned above also includes the cost of accessing the data, in the same way as the actual benchmark.

However, because the overhead is measured in a single process, the cost is typically the cost with "hot" caches. In the Figure 2, each size is plotted as a line, with context switch times on the Y axis, number of processes on the X axis, and the process size as the data set. The process size and the hot cache overhead costs for the pipe read/writes and any data access is what is labeled as size=0KB overhead=10. The size is in kilobytes and the overhead is in microseconds.

The context switch time does not include anything other than the context switch, provided that all the benchmark processes fit in the cache. If the total size of all of the benchmark processes is larger than the cache size, the cost of each context switch will include cache misses. We are trying to show realistic context switch times as a function of both size and number of processes.

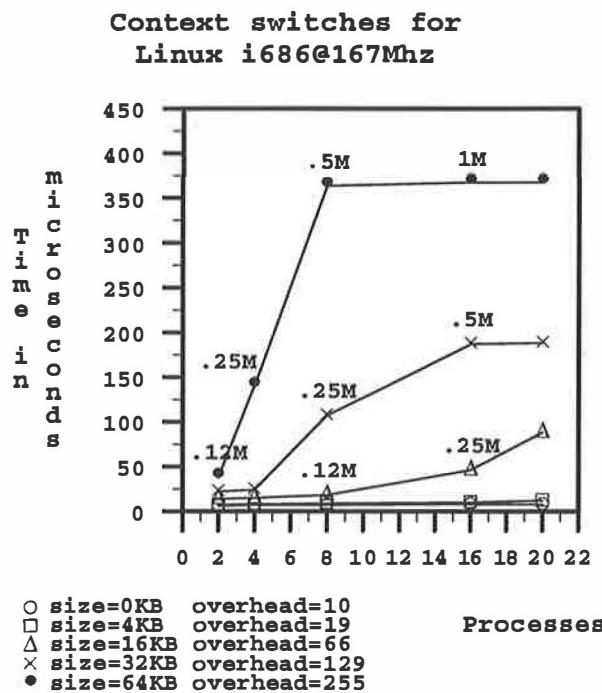


Figure 2. Context switch times

Results for an Intel Pentium Pro system running Linux at 167 MHz are shown in Figure 2. The data points on the figure are labeled with the working set due to the sum of data in all of the processes. The actual working set is larger, as it includes the process and kernel overhead as well. One would expect the context switch times to stay constant until the working set is approximately the size of the second level cache. The Intel system has a 256K second level cache, and the context switch times stay almost constant until about 256K (marked as .25M in the graph).

⁴ All arrays are at the same virtual address in all processes.

- **Cache issues** The context switch benchmark is a deliberate measurement of the effectiveness of the caches across process context switches. If the cache does not include the process identifier (PID, also sometimes called an address space identifier) as part of the address, then the cache must be flushed on every context switch. If the cache does not map the same virtual addresses from different processes to different cache lines, then the cache will appear to be flushed on every context switch.

If the caches do not cache across context switches there would be no grouping at the lower left corner of Figure 2, instead, the graph would appear as a series of straight, horizontal, parallel lines. The number of processes will not matter, the two process case will be just as bad as the twenty process case since the cache would not be useful across context switches.

System	2 processes		8 processes	
	0KB	32KB	0KB	32KB
Linux/i686	6	18	7	101
Linux/i586	10	163	13	215
Linux/Alpha	11	70	13	78
IBM Power2	13	16	18	43
Sun Ultra1	14	31	20	102
DEC Alpha@300	14	17	22	41
IBM PowerPC	16	87	26	144
HP K210	17	17	18	99
Unixware/i686	17	17	18	72
FreeBSD/i586	27	34	33	102
Solaris/i686	36	54	43	118
SGI Indigo2	40	47	38	104
DEC Alpha@150	53	68	59	134
SGI Challenge	63	80	69	93
Sun SC1000	107	142	104	197

Table 10. Context switch time (microseconds)

We picked four points on the graph and extracted those values for Table 10. The complete set of values, as well as tools to graph them, are included with lmbench.

Note that multiprocessor context switch times are frequently more expensive than uniprocessor context switch times. This is because multiprocessor operating systems tend to have very complicated scheduling code. We believe that multiprocessor context switch times can be, and should be, within 10% of the uniprocessor times.

Linux does quite well on context switching, especially on the more recent architectures. By comparing the Linux 2 0K processes to the Linux 2 32K processes, it is apparent that there is something wrong with the Linux/i586 case. If we look back to Table 6, we can find at least part of the cause. The second level cache latency for the i586 is substantially worse than either the i686 or the Alpha.

Given the poor second level cache behavior of the PowerPC, it is surprising that it does so well on context switches, especially the larger sized cases.

The Sun Ultra1 context switches quite well in part because of enhancements to the register window handling in SPARC V9.

6.7. Interprocess communication latencies

Interprocess communication latency is important because many operations are control messages to another process (frequently on another system). The time to tell the remote process to do something is pure overhead and is frequently in the critical path of important functions such as distributed applications (e.g., databases, network servers).

The interprocess communication latency benchmarks typically have the following form: pass a small message (a byte or so) back and forth between two processes. The reported results are always the microseconds needed to do one round trip. For one way timing, about half the round trip is right. However, the CPU cycles tend to be somewhat asymmetric for one trip: receiving is typically more expensive than sending.

- **Pipe latency.** Unix pipes are an interprocess communication mechanism implemented as a one-way byte stream. Each end of the stream has an associated file descriptor; one is the write descriptor and the other the read descriptor.

Pipes are frequently used as a local IPC mechanism. Because of the simplicity of pipes, they are frequently the fastest portable communication mechanism.

Pipe latency is measured by creating a pair of pipes, forking a child process, and passing a word back and forth. This benchmark is identical to the two-process, zero-sized context switch benchmark, except that it includes both the context switching time and the pipe overhead in the results. Table 11 shows the round trip latency from process A to process B and back to process A.

System	Pipe latency
Linux/i686	26
Linux/i586	33
Linux/Alpha	34
Sun Ultra1	62
IBM PowerPC	65
Unixware/i686	70
DEC Alpha@300	71
HP K210	78
IBM Power2	91
Solaris/i686	101
FreeBSD/i586	104
SGI Indigo2	131
DEC Alpha@150	179
SGI Challenge	251
Sun SC1000	278

Table 11. Pipe latency (microseconds)

The time can be broken down to two context switches plus four system calls plus the pipe overhead.

The context switch component is two of the small processes in Table 10. This benchmark is identical to the context switch benchmark in [Ousterhout90].

- **TCP and RPC/TCP latency.** TCP sockets may be viewed as an interprocess communication mechanism similar to pipes with the added feature that TCP sockets work across machine boundaries.

TCP and RPC/TCP connections are frequently used in low-bandwidth, latency-sensitive applications. The default Oracle distributed lock manager uses TCP sockets, and the locks per second available from this service are accurately modeled by the TCP latency test.

System	TCP	RPC/TCP
Linux/i686	216	346
Sun Ultra1	162	346
DEC Alpha@300	267	371
FreeBSD/i586	256	440
Solaris/i686	305	528
Linux/Alpha	429	602
HP K210	146	606
SGI Indigo2	278	641
IBM Power2	332	649
IBM PowerPC	299	698
Linux/i586	467	713
DEC Alpha@150	485	788
SGI Challenge	546	900
Sun SC1000	855	1386

Table 12. TCP latency (microseconds)

Sun's RPC is layered either over TCP or over UDP. The RPC layer is responsible for managing connections (the port mapper), managing different byte orders and word sizes (XDR), and implementing a remote procedure call abstraction. Table 12 shows the same benchmark with and without the RPC layer to show the cost of the RPC implementation.

TCP latency is measured by having a server process that waits for connections and a client process that connects to the server. The two processes then exchange a word between them in a loop. The latency reported is one round-trip time. The measurements in Table 12 are local or loopback measurements, since our intent is to show the overhead of the software. The same benchmark may be, and frequently is, used to measure host-to-host latency.

Note that the RPC layer frequently adds hundreds of microseconds of additional latency. The problem is not the external data representation (XDR) layer — the data being passed back and forth is a byte, so there is no XDR to be done. There is no justification for the extra cost; it is simply an expensive implementation. DCE RPC is worse.

- **UDP and RPC/UDP latency.** UDP sockets are an alternative to TCP sockets. They differ in that UDP sockets are unreliable messages that leave the retransmission issues to the application. UDP sockets have a

System	UDP	RPC/UDP
Linux/i686	93	180
Sun Ultra1	197	267
Linux/Alpha	180	317
DEC Alpha@300	259	358
Linux/i586	187	366
FreeBSD/i586	212	375
Solaris/i686	348	454
IBM Power2	254	531
IBM PowerPC	206	536
HP K210	152	543
SGI Indigo2	313	671
DEC Alpha@150	489	834
SGI Challenge	678	893
Sun SC1000	739	1101

Table 13. UDP latency (microseconds)

few advantages, however. They preserve message boundaries, whereas TCP does not; and a single UDP socket may send messages to any number of other sockets, whereas TCP sends data to only one place.

UDP and RPC/UDP messages are commonly used in many client/server applications. NFS is probably the most widely used RPC/UDP application in the world.

Like TCP latency, UDP latency is measured by having a server process that waits for connections and a client process that connects to the server. The two processes then exchange a word between them in a loop. The latency reported is round-trip time. The measurements in Table 13 are local or loopback measurements, since our intent is to show the overhead of the software. Again, note that the RPC library can add hundreds of microseconds of extra latency.

System	Network	TCP latency	UDP latency
Sun Ultra1	100baseT	280	308
FreeBSD/i586	100baseT	365	304
HP 9000/735	fddi	425	441
SGI Indigo2	10baseT	543	602
HP 9000/735	10baseT	592	603
SGI PowerChallenge	hippi	1068	1099
Linux/i586@90Mhz	10baseT	2954	1912

Table 14. Remote latencies (microseconds)

- **Network latency.** We have a few results for over the wire latency included in Table 14. As might be expected, the most heavily used network interfaces (i.e., ethernet) have the lowest latencies. The times shown include the time on the wire, which is about 130 microseconds for 10Mbit ethernet, 13 microseconds for 100Mbit ethernet and FDDI, and less than 10 microseconds for Hippi.

- **TCP connection latency.** TCP is a connection-based, reliable, byte-stream-oriented protocol. As part of this reliability, a connection must be established before any data can be transferred. The connection is

accomplished by a “three-way handshake,” an exchange of packets when the client attempts to connect to the server.

Unlike UDP, where no connection is established, TCP sends packets at startup time. If an application creates a TCP connection to send one message, then the startup time can be a substantial fraction of the total connection and transfer costs. The benchmark shows that the connection cost is approximately half of the cost.

Connection cost is measured by having a server, registered using the port mapper, waiting for connections. The client figures out where the server is registered and then repeatedly times a connect system call to the server. The socket is closed after each connect. Twenty connects are completed and the fastest of them is used as the result. The time measured will include two of the three packets that make up the three way TCP handshake, so the cost is actually greater than the times listed.

System	TCP connection
HP K210	238
Linux/i686	263
IBM Power2	339
FreeBSD/i586	418
Linux/i586	606
SGI Indigo2	667
SGI Challenge	716
Sun Ultra1	852
Solaris/i686	1230
Sun SC1000	3047

Table 15. TCP connect latency (microseconds)

Table 15 shows that if the need is to send a quick message to another process, given that most packets get through, a UDP message will cost a send and a reply (if positive acknowledgments are needed, which they are in order to have an apples-to-apples comparison with TCP). If the transmission medium is 10Mbit Ethernet, the time on the wire will be approximately 65 microseconds each way, or 130 microseconds total. To do the same thing with a short-lived TCP connection would cost 896 microseconds of wire time alone.

The comparison is not meant to disparage TCP; TCP is a useful protocol. Nor is the point to suggest that all messages should be UDP. In many cases, the difference between 130 microseconds and 900 microseconds is insignificant compared with other aspects of application performance. However, if the application is very latency sensitive and the transmission medium is slow (such as serial link or a message through many routers), then a UDP message may prove cheaper.

6.8. File system latency

File system latency is defined as the time required to create or delete a zero length file. We define it this way because in many file systems, such as the BSD fast file system, the directory operations are done synchronously in order to maintain on-disk integrity. Since the file data is typically cached and sent to disk at some later date, the file creation and deletion become the bottleneck seen by an application. This bottleneck is substantial: to do a synchronous update to a disk is a matter of tens of milliseconds. In many cases, this bottleneck is much more of a perceived performance issue than processor speed.

The benchmark creates 1,000 zero-sized files and then deletes them. All the files are created in one directory and their names are short, such as "a", "b", "c", ... "aa", "ab",

System	FS	Create	Delete
Linux/i686	EXT2FS	751	45
HP K210	HFS	579	67
Linux/i586	EXT2FS	1,114	95
Linux/Alpha	EXT2FS	834	115
Unixware/i686	UFS	450	369
SGI Challenge	XFS	3,508	4,016
DEC Alpha@300	ADVFS	4,255	4,184
Solaris/i686	UFS	23,809	7,246
Sun Ultra1	UFS	18,181	8,333
Sun SC1000	UFS	25,000	11,111
FreeBSD/i586	UFS	28,571	11,235
SGI Indigo2	EFS	11,904	11,494
DEC Alpha@150	?	38,461	12,345
IBM PowerPC	JFS	12,658	12,658
IBM Power2	JFS	13,333	12,820

Table 16. File system latency (microseconds)

The create and delete latencies are shown in Table 16. Notice that Linux does extremely well here, 2 to 3 orders of magnitude faster than the slowest systems. However, Linux does not guarantee anything about the disk integrity; the directory operations are done in memory. Other fast systems, such as SGI's XFS, use a log to guarantee the file system integrity. The slower systems, all those with ~10 millisecond file latencies, are using synchronous writes to guarantee the file system integrity. Unless Unixware has modified UFS substantially, they must be running in an unsafe mode since the FreeBSD UFS is much slower and both file systems are basically the 4BSD fast file system.

6.9. Disk latency

Included with lmbench is a small benchmarking program useful for measuring disk and file I/O. lndd, which is patterned after the Unix utility dd, measures both sequential and random I/O, optionally generates patterns on output and checks them on input, supports flushing the data from the buffer cache on systems that support msync, and has a very flexible user interface. Many I/O benchmarks can be

trivially replaced with a perl script wrapped around lmdd.

While we could have generated both sequential and random I/O results as part of this paper, we did not because those benchmarks are heavily influenced by the performance of the disk drives used in the test. We intentionally measure only the system overhead of a SCSI command since that overhead may become a bottleneck in large database configurations.

Some important applications, such as transaction processing, are limited by random disk IO latency. Administrators can increase the number of disk operations per second by buying more disks, until the processor overhead becomes the bottleneck. The lmdd benchmark measures the processor overhead associated with each disk operation, and it can provide an upper bound on the number of disk operations the processor can support. It is designed for SCSI disks, and it assumes that most disks have 32-128K read-ahead buffers and that they can read ahead faster than the processor can request the chunks of data.⁵

The benchmark simulates a large number of disks by reading 512byte transfers sequentially from the raw disk device (raw disks are unbuffered and are not read ahead by Unix). Since the disk can read ahead faster than the system can request data, the benchmark is doing small transfers of data from the disk's track buffer. Another way to look at this is that the benchmark is doing memory-to-memory transfers across a SCSI channel. It is possible to generate loads of more than 1,000 SCSI operations/second on a single SCSI disk. For comparison, disks under database load typically run at 20-80 operations per second.

System	Disk latency
SGI Challenge	920
SGI Indigo2	984
HP K210	1103
DEC Alpha@150	1436
Sun SC1000	1466
Sun Ultra	2242

Table 17. SCSI I/O overhead (microseconds)

The resulting overhead number represents a lower bound on the overhead of a disk I/O. The real overhead numbers will be higher on SCSI systems because most SCSI controllers will not disconnect if the request can be satisfied immediately. During the benchmark, the processor simply sends the request and transfers the data, while during normal operation, the processor will send the request, disconnect, get

⁵ This may not always be true: a processor could be fast enough to make the requests faster than the rotating disk. If we take 6M/second to be disk speed, and divide that by 512 (the minimum transfer size), that is 12,288 I/Os/second, or 81 microseconds/IO. We don't know of any processor/OS/IO controller combinations that can do an IO in 81 microseconds.

interrupted, reconnect, and transfer the data.

This technique can be used to discover how many drives a system can support before the system becomes CPU-limited because it can produce the overhead load of a fully configured system with just a few disks.

7. Future work

There are several known improvements and extensions that could be made to lmbench.

- **Memory latency.** The current benchmark measures clean-read latency. By clean, we mean that the cache lines being replaced are highly likely to be unmodified, so there is no associated write-back cost. We would like to extend the benchmark to measure dirty-read latency, as well as write latency. Other changes include making the benchmark impervious to sequential prefetching and measuring TLB miss cost.

- **MP benchmarks.** None of the benchmarks in lmbench is designed to measure any multiprocessor features directly. At a minimum, we could measure cache-to-cache latency as well as cache-to-cache bandwidth.

- **Static vs. dynamic processes.** In the process creation section, we allude to the cost of starting up processes that use shared libraries. When we figure out how to create statically linked processes on all or most systems, we could quantify these costs exactly.

- **McCalpin's stream benchmark.** We will probably incorporate part or all of this benchmark into lmbench.

- **Automatic sizing.** We have enough technology that we could determine the size of the external cache and autosize the memory used such that the external cache had no effect.

- **More detailed papers.** There are several areas that could yield some interesting papers. The memory latency section could use an in-depth treatment, and the context switching section could turn into an interesting discussion of caching technology.

8. Conclusion

lmbench is a useful, portable micro-benchmark suite designed to measure important aspects of system performance. We have found that a good memory subsystem is at least as important as the processor speed. As processors get faster and faster, more and more of the system design effort will need to move to the cache and memory subsystems.

9. Acknowledgments

Many people have provided invaluable help and insight into both the benchmarks themselves and the paper. The USENIX reviewers were especially helpful. We thank all of them and especially thank: Ken Okin (SUN), Kevin Normoyle (SUN), Satya Nishtala (SUN),

Greg Chesson (SGI), John Mashey (SGI), Neal Nuckles (SGI), John McCalpin (Univ. of Delaware), Ron Minnich (Sarnoff), Chris Ruemmler (HP), Tom Rokicki (HP), and John Weitz (Digidesign).

We would also like to thank all of the people that have run the benchmark and contributed their results; none of this would have been possible without their assistance.

Our thanks to all of the free software community for tools that were used during this project. lmbench is currently developed on Linux, a copylefted Unix written by Linus Torvalds and his band of happy hackers. This paper and all of the lmbench documentation was produced using the groff suite of tools written by James Clark. Finally, all of the data processing of the results is done with perl written by Larry Wall.

Sun Microsystems, and in particular Paul Borrill, supported the initial development of this project. Silicon Graphics has supported ongoing development that turned into far more time than we ever imagined. We are grateful to both of these companies for their financial support.

10. Obtaining the benchmarks

The benchmarks are available at http://reality.sgi.com/employees/lm_engr/lmbench.tgz as well as via a mail server. You may request the latest version of lmbench by sending email to archives@slovaix.engr.sgi.com with *lmbench-current** as the subject.

References

- [Chen94] P. M. Chen and D. A. Patterson, "A new approach to I/O performance evaluation – self-scaling I/O benchmarks, predicted I/O performance," *Transactions on Computer Systems*, 12 (4), pp. 308-339, November 1994.
- [Chen93] Peter M. Chen and David Patterson, "Storage performance – metrics and benchmarks," *Proceedings of the IEEE*, 81 (8), pp. 1151-1165, August 1993.
- [Fenwick95] David M. Fenwick, Denis J. Foley, William B. Gist, Stephen R. VanDoren, and Daniel Wissell, "The AlphaServer 8000 series: high-end server platform development," *Digital Technical Journal*, 7 (1), pp. 43-65, August 1995.
- [Hennessy96] John L. Hennessy and David A. Patterson, "Computer Architecture A Quantitative Approach, 2nd Edition," Morgan Kaufman, 1996.
- [McCalpin95] John D. McCalpin, "Memory bandwidth and machine balance in current high performance computers," *IEEE Technical Committee on Computer Architecture newsletter*, to appear, December 1995.
- [McVoy91] L. W. McVoy and S. R. Kleiman, "Extent-like Performance from a Unix File System," pp. 33-43, Proceedings USENIX Winter Conference, January 1991.
- [Ousterhout90] John K. Ousterhout, "Why aren't operating systems getting faster as fast as hardware?" pp. 247-256, Proceedings USENIX Summer Conference, June 1990.
- [Park90] Arvin Park and J. C. Becker, "IOStone: a synthetic file system benchmark," *Computer Architecture News*, 18 (2), pp. 45-52, June 1990.
- [Saavedra95] R.H. Saavedra and A.J. Smith, "Measuring cache and TLB performance and their effect on benchmark runtimes," *IEEE Transactions on Computers*, 44 (10), pp. 1223-1235, October 1995.
- [Stallman89] Free Software Foundation, Richard Stallman, "General Public License," 1989. Included with lmbench.
- [Toshiba94] Toshiba, "DRAM Components and Modules," pp. A59-A77,C37-C42, Toshiba America Electronic Components, Inc., 1994.
- [Wolman89] Barry L. Wolman and Thomas M. Olson, "IOBENCH: a system independent IO benchmark," *Computer Architecture News*, 17 (5), pp. 55-70, September 1989.

Biographical information

Larry McVoy currently works Silicon Graphics in the Networking Systems Division on high performance networked file systems and networking architecture. His computer interests include hardware architecture, software implementation and architecture, performance issues, and free (GPLed) software issues. Previously at Sun, he was the architect for the SPARC Cluster product line, redesigned and wrote an entire source management system (now productized as TeamWare), implemented UFS clustering, and implemented all of the Posix 1003.1 support in SunOS 4.1. Concurrent with Sun work, he lectured at Stanford University on Operating Systems. Before Sun, he worked on the ETA systems supercomputer Unix port. He may be reached by electronically via lm@sgi.com or by phone at (415) 933-1804.

Carl Staelin works for Hewlett-Packard Laboratories in the External Research Program. His research interests include network information infrastructures and high performance storage systems. He worked for HP at U.C. Berkeley on the 4.4BSD LFS port, the HighLight hierarchical storage file system, the Mariposa distributed database, and the NOW project. He received his PhD in Computer Science from Princeton University in 1991 in high performance file system design. He may be reached electronically via staelin@hpl.hp.com.

Process-labeled kernel profiling: a new facility to profile system activities

Shingo Nishioka, Atsuo Kawaguchi, and Hiroshi Motoda

Advanced Research Laboratory, Hitachi Ltd.

Abstract

Profiling tools that empirically measure the resource usage of a program have been widely used in program development. These tools have traditionally focused on the behavior of the target program. The target program, however, actually performs its job in collaboration with other programs, such as servers and an operating system kernel, in a modern system environment. Process-labeled kernel profiling is a novel facility that measures and attributes the kernel resource consumption of programs benefiting from it. This facility, in conjunction with a conventional profiler, enables a programmer to grasp the resource consumption of programs from an overall system point of view. Using this information, the programmer is better able to reduce overall resource consumption.

1. Introduction

Profiling is one of a number of important techniques used to improve the performance of a complex program. Many tools for profiling, such as prof[1], gprof[2], and cprof[3], have been developed and widely used in application program development. A programmer uses these tools to measure the resource usage (e.g. CPU time) of a program empirically, and to analyze where and why it consumes resources during execution. The measurement results provide clues to finding critical portions of the program which affect its performance, and they also provide evidence that the programmer's code modifications actually improve program performance.

A typical example of complex programs is an operating system kernel. Profiling is useful not only in improving user programs, but also in improving and tuning kernel codes. By empirically measuring resource usage in the kernel over a sufficiently long period of time, a kernel developer is able to locate problems affecting the overall performance of a

system. Some problems may need kernel code modification, while others can be eased by changing kernel parameters[4].

User program profiling and kernel profiling are used independently since the kernel and user program are separate. One of the kernel's important roles is offering a well-defined, abstract, and convenient program interface that hides the system's internal operations. Consequently, programmers are able to develop a portable, clearly organized application program without any special knowledge of these internal operations. Moreover, expert programmers understand these operations better than novices, and develop programs that access the program interface in a manner that can be efficiently processed by the kernel. Such implicit optimization often brings about better performance than expected. Providing kernel profiling data as well as user program profiling results should help expert programmers to write more efficient codes from an overall system point of view.

Recent operating systems have relied on many user-level programs (servers) to provide system services. The micro-kernel architecture is a typical example of such an approach. Providing both user programs and kernel profiling data simultaneously is extremely useful in the development operating systems to establish system structures, define server interfaces, write codes, tune prototypes, and so on.

Process-labeled kernel profiling (pkprof) is a novel profiling facility that measures which resources are used in an operating system kernel by which user program (process¹). It keeps track of which process a kernel is serving, and labels particular resource consumption for a particular process that is to benefit by that consumption. Utility tools are provided to control measurements, to manage profile data, and to

¹We use the word 'process' to represent an execution thread that a kernel identifies as an object of system resource allotment. The UNIX process created by the `fork` system call is such an object.

analyze data.

Pkprof, in conjunction with a user level profiler, enables a programmer to understand a program's behavior in both the user and kernel space. It reveals hidden system activities that the programmer is unaware of. Such information is extremely beneficial to both the application program and operating system development.

2. Pkprof: Process-labeled kernel profiling

An operating system kernel offers a set of system calls to provide services to user programs. The services are implemented as kernel routines that run in the kernel space. A user program triggers execution of the routines by issuing a system call. Resources consumed by the execution, therefore, can be attributed to that program.

A kernel handles external asynchronous events. Resources consumed to handle such events are also attributed to programs benefiting from consumption. For example, when a network packet arrives, network interface hardware interrupts the normal flow of program execution and invokes the kernel's interrupt handling routines. The interrupt handling routines copy the received packet to a network data buffer and the system then resumes normal execution. The received packet is eventually read by a user program. Consequently, resources consumed by packet reception are attributed to that program when it turns out to have benefited from the reception.

A process-labeled kernel profiler monitors kernel activities and reports where and for what process (program) a kernel consumes resources such as CPU time. In other words, it measures the kernel resources consumed by:

- processes that request kernel services, and
- asynchronous events that are later attributed to processes.

It then reports the extent to which each process benefits from the use of these resources.

Figure 2.1 shows a report on process-labeled kernel profiling. Figure 2.1(a) shows flat profiling results of a kernel running over a period of time. Each line represents how much time is spent by the kernel routine. `we_bcopy_in()`, for example, was called 6622 times (`calls` column) and ran for 4.33 seconds (`self seconds`); that accounts for 4.6% of the total time that elapsed in the kernel (`% time`) during measurement. Each call took 0.65 ms (`self ms/call`) within `we_bcopy_in()` itself and took 0.65 ms (`total ms/call`) for itself and its descendant functions on average. Figure 2.1(b) shows the same information with respect to a process

time	cumulative	self	calls	self ms/call	total ms/call	name
65.7	62.24	62.24	595	104.61	104.61	137k:idle [4]
7.5	69.35	7.11	5930	1.20	1.20	137k:in_cksum [7]
4.6	73.68	4.33	6622	0.65	0.65	137k:we_bcopy_in [11]
4.3	77.74	4.06				137k:_mcOUNT [1656]
2.8	80.36	2.62	2923	0.90	0.90	137k:rpcPutc [17]
2.5	82.76	2.40	30766	0.08	0.08	137k:ovbcopy [22]
2.2	84.80	2.04	190	10.74	10.74	137k:blkclr [25]

(a): Example listing of kernel flat profiling

time	cumulative	self	calls	self ms/call	total ms/call	name
24.2	2.62	2.62	2620	1.00	1.00	137k:in_cksum [9]
13.7	4.10	1.48	12864	0.12	0.12	137k:ovbcopy [15]
12.7	5.48	1.38				137k:_mcOUNT [1656]
12.5	6.84	1.36	3094	0.44	0.44	137k:we_bcopy_in [16]
5.0	7.38	0.54	62606	0.01	0.01	137k:_splx [25]
2.6	8.14	0.28	7877	0.04	0.07	137k:malloc <cycle 1> [24]
2.4	8.40	0.26	8888	0.03	0.04	137k:free [36]

(b): CPU time usage of the kernel for the cat program.

Figure 2.1: Example report of pkprof.

index	etime	self	descendents	called/total	called+self	parents	name	children
[8]	30.2	0.00	2.86	65/65	65/65	137u:raw_cat [7]		
		0.00	2.86	65	65	137u:read [8]		137k:read [9]
		0.00	2.86	65/65	65/65			
[9]	30.2	0.00	2.86	65/65	65/65	137u:read [8]		
		0.00	2.86	65	65	137k:read [9]		137k:vn_read [10]
		0.01	2.85	65/65	65/65			
[10]	30.2	0.01	2.85	65/65	65/65	137k:read [9]		
		0.01	2.85	65	65	137k:vn_read [10]		
		0.00	2.84	65/76	65/76	137k:nfs_read [2]		
		0.00	0.02	65/163	65/163	137k:lease_check [90]		
		0.00	0.00	65/105	65/105	137k:nfs_lock [282]		
		0.00	0.00	65/101	65/101	137k:nfs_unlock [283]		

Figure 2.2: Example listing of seamless profiling.

executing a `cat` program over the same period. That is, it shows how much CPU time² is consumed by each kernel routine in processing `cat`'s requests. A process identifier and symbol 'k' are prefixed to the name for each function, e.g. `137k:in_cksum[9]`. The '137' denotes the process identifier of the `cat` process and the 'k' denotes that the function is the kernel's. We can see that some kernel routines such as `ovbcopy()` are mainly executed for the `cat` process.

A typical pkprof application is *seamless profiling*, which simultaneously measures the resource consumption of a program both in the user and kernel space. A listing of seamless profiling is shown in Figure 2.2. The listing is shown in the same format that `gprof`[2] generates except that the process identifier and symbols 'k' or 'u' have been prefixed to the name of each function, e.g. `137k:read[9]`. The '137' denotes the process identifier and 'k' denotes that the function is the kernel's. Symbol 'u' is shown if the function is the user program's. The listing shows,

² It can be said that a kernel profiler measures the elapsed time for each kernel routine basically, because the profiler usually measures how much time has been spent in the idle routine as well as other kernel routines. Attributing the idle routine to processes, in general, is a very difficult task. We have decided to attribute the idle routine only to the kernel itself. As kernel profiling results in respect to a process, it can be said to deal with the CPU time usage of kernel routines invoked by that particular process.

index[8] for example, that the process 137's `read()` function ran for 30.2% of the total CPU time during measurement (`% time` column). It used 0.00 seconds within `137u:read()` itself (`self`), and the descendants of `137u:read()`, i.e. `137k:read()` in this case, needed 2.86 seconds to execute in CPU time. Programmers can grasp the impact on the program of kernel CPU usage from the listing, as well as the CPU time usage of the program itself. They may be able to lower the total CPU time usage by changing the program algorithm, the system calls it issues, or their arguments.

Pkprof is applicable to a group of processes. Programmers can examine the kernel usage and impact on system performance of processes as well as each single process. They may be able to improve the overall performance of the system by carefully examining the listings, finding actual performance bottlenecks, and fixing them.

Applying pkprof to a group of processes is particularly useful in programming based on a client-server scheme. To improve performance in this type of scheme it is not sufficient to profile only the client program since a great deal of work for a job is done by the server. Profiling the server may often be sufficient to improve performance, but applying pkprof to the server and clients may help programmers dramatically improve performance through: changing the scheme itself, changing shared work between the server and clients, changing communication methods, or changing the protocols used for communication.

The pkprof facility offers the following utilities to programmers:

- turns profiler on and off,
- specifies processes the profiler monitors,
- specifies resources to profile,
- reports on measurements.

3. Basic structure

Pkprof monitors the resource consumption in a kernel, attributes consumption to the process benefiting from it, and records consumption with attribution. Pkprof uses the same techniques to monitor resource consumption as conventional profiling tools; it uses embedding codes and sampling techniques. Pkprof switches a profile data buffer when the kernel switches a process or serves asynchronous events (interrupts). It tries to determine what process will later benefit from each asynchronous event.

3.1 Attributing resource consumption

Sources of resource consumption in a modern, multi-tasking operating system kernel can be classified into the following two types of events:

- events caused by a process running, and
- events caused by a device.

The former occur synchronously in respect to the execution of the process. The process asks for kernel services explicitly (e.g. system calls) or implicitly (e.g. page faults) through the events. Therefore, consumption attribution is performed by keeping track of process switching in the kernel. The process running currently is the one being served by the kernel. The latter, however, occurs asynchronously and often has nothing to do with the running process.

The asynchronous event is usually only partially processed. Partial processing results are stored in a temporal buffer, and the remaining processing is performed when the process makes a request that requires response from the processing results. Therefore, resource consumption attribution caused by the event must be deferred until the process that benefits from the event is identified.

Assigning a unique identifier to every asynchronous event is one technique to handle the situation above. The identifier serves as a classification tag and records the resource consumed by processing of the asynchronous event. Once a process benefits from the tagged resource, all the resource consumption recorded with the same tag is attributed to that process.

The address of a data buffer allocated to handle an asynchronous event can be used as such an identifier. The buffer is usually passed from routine to routine during partial processing. The address, therefore, can be used to tag the resource consumed by processing. Note that the asynchronous event handling routines may allocate another buffer and use this during partial processing. The address of a newly allocated buffer should also be treated as another identifier in such cases.

3.2 Recording profile data

The pkprof logically allocates a profile log buffer for every profiled process. It also allocates a buffer for every asynchronous event. Access routines are provided for profiler routines so that they can efficiently record resource consumption in the buffers. It is preferable to construct buffer structures dynamically, and on demand because the number of processes to be profiled is not known in advance. Static allocation of buffers may restrict profile measurements and impose a performance penalty while the profiling facility is turned off.

It should be noted that profile data buffers for asynchronous events are allocated and de-allocated much more frequently than those for profiled processes. When a process that benefits from an

asynchronous event is found, the contents of the data buffer for that event can be merged into the profile data buffer for the found process. The buffer for the asynchronous event, then, can be de-allocated and reused for another event. Quick allocation and reuse of the buffers are the keys to making asynchronous event attribution more practical.

3.3 Implementing pkprof

Implementation of a pkprof facility, unlike conventional kernel profiles, heavily depends on the structures of a target operating system. It requires higher-level knowledge of the target such as

- events caused by a process,
- handling of synchronous events,
- location and manner of process switching,
- the way asynchronous events are caught and processed,
- data structures used to process events, and so on.

The part in pkprof concerned with synchronous events can be implemented on top of a conventional kernel profiling facility[4][5]. Since served process tracking and profile log buffer management are functionally independent of a conventional profiler, we can add both functions without modifying the existing profiler. Once the functions are added, we can modify existing profiler routines so that they refer to the currently served process and record resource consumption properly. Of course, we need to adapt existing utilities to the new profile facility.

Implementation of the other part, i.e. the attribution of asynchronous events, needs much more modification of the target operating system. It generally includes:

- enumerating places where asynchronous events are caught and handled,
- embedding codes into places that assign identifiers and profile data buffers to events,
- embedding codes, where appropriate, to track asynchronous event handling, and
- embedding codes to resolve process-event correspondence and to merge event profile data into process profile data.

Note that asynchronous event processing occupies a relatively small fraction of kernel CPU usage in some environments. Implementing only the attribution of synchronous events might prove adequate in such cases.

4. Implementation

This section describes sample implementation of pkprof on a 4.4BSD-Lite UNIX operating system. After this, we will refer to sample implementation as PKPROF. Although PKPROF is experimental and

incomplete, it produces the least functionality that is needed to evaluate pkprof. Many techniques used for sample implementation, such as monitor, sampler, and profile data management, are independent of 4.4BSD-Lite kernel architecture. They should be applied to the implementation of pkprof on other operating system kernels. Descriptions of the process switch and asynchronous event tracking, however, depend heavily on the design of the 4.4BSD-Lite kernel.

4.1 Overview

The 4.4BSD-Lite kernel offers a configurable kernel profiling facility called KPROF. KPROF measures:

- how often each kernel routine is called, and
- how much time is spent executing each kernel routine.

The former is implemented by inserting monitoring codes (“monitor”) in the prologue for each kernel routine so that the monitor is executed during every invocation of the routine. The latter is actually measured by sampling the address at which the CPU is executing an instruction. Execution times for kernel routines are inferred from the distribution of the samples within the total execution time of the kernel. This implementation scheme used by KPROF is the same scheme used by gprof.

PKPROF is an extension of KPROF; the monitor and sampler scheme is basically the same as that of gprof. We have added and modified KPROF codes so that the monitor and sampler are able to classify the profile data, and store them in the per-process and per-event profile data buffers properly. PKPROF stores profile data into buffers in the kernel memory space.

Major issues on implementation include:

- where the kernel switches one running process to another,
- how asynchronous events are tagged and tracked,
- how the monitor and sampler keep track of process switching and event processing,
- how profile data buffers are allocated and organized, and
- how measurements are controlled and how profile data is accessed from outside the kernel.

Limiting the impact of performance was one of the key criteria in designing the mechanism and data structures of PKPROF (Section 5.1 discusses overhead).

The 4.4BSD-Lite operating system enters into the kernel mode when one of the following events occur:

1. a process issues a system call.

2. a process accesses an address to which a physical memory page is not assigned at the time (i.e. page fault).
3. a device interrupts the execution of the current process.

When the first event occurs, the kernel starts processing the process's request immediately. If a requisite resource is unavailable at that time, the kernel switches the process to another process. Similar processing also occurs for the second event. The third event, however, starts processing which may be irrelevant to the current process. For example, a process may be interrupted to handle the arrival of a network packet even if this process never accesses the network.

We have added two global variables; `svdproc` indicates what process the kernel is serving and `svdevent` indicates what event the kernel is handling. The kernel updates the value of each variable when process switching and an asynchronous event occur. The monitor and the sampler refer to the variables and store profile data to the proper profile data buffer. Profile data for an asynchronous event handling are merged into the profile data buffer for a process when that process is found to benefit from event handling.

4.2 Served process tracking

The 4.4BSD-Lite kernel maintains a global variable named `curproc`. This points to a *proc* structure that contains the state of the currently running process. Although the name stands for "current process", the value of `curproc` cannot be used to track the "current served process", because its value often remains unchanged while the kernel is serving other processes. For example, the interrupt handler never changes its value. The value of `curproc` is also left untouched while the kernel does system-wide jobs such as process priority calculations.

We have added a new variable named `svdproc` (stands for "served process") that shows a process the kernel is actually serving. Codes that we have embedded into the context switching related kernel routines update the value of `svdproc`.

Figure 4.1 shows the call flow diagram for kernel routines that are responsible for the execution of context switching. Each node shows the kernel routine and each solid arrow indicates the caller-callee relationship between the routines. For example, the figure shows that `exit1()` calls `cpu_exit()`. Each gray arrow shows the transition from the user mode to the kernel mode. The `interrupt()` is invoked by hardware interrupts such as interval timer clock, disk, and network interrupts.

Only kernel routines marked with '*' switch the

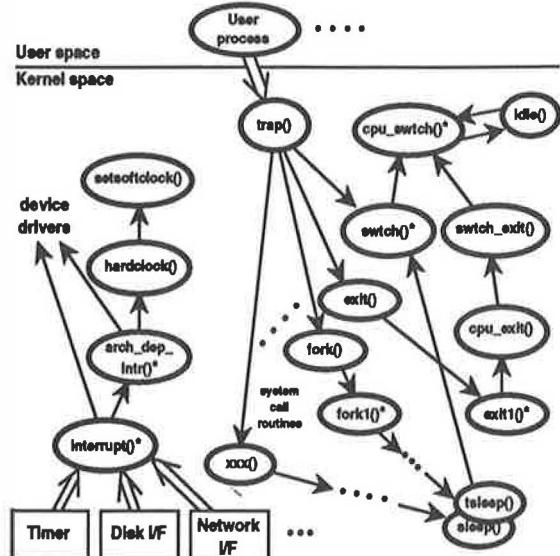


Figure 4.1: Call flow of kernel routines.

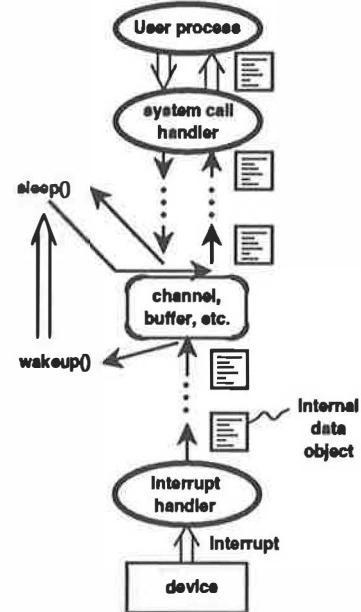


Figure 4.2: Asynchronous event processing.

currently served process. `Switch()` switches the current process to another. `Exit1()` handles process termination. The value of `svdproc` is set to NULL when the process terminates, which means no process is served by the kernel. `Interrupt()` also changes `svdproc` to NULL because it invokes the kernel's interrupt handling activities.

4.3 Asynchronous event tracking

Figure 4.2 depicts a generic view of asynchronous event processing for the 4.4BSD-Lite kernel. Device interrupt is caught by the lowest interrupt handler. It then invokes an appropriate interrupt handling routine

for the device driver. The interrupt handling routine and other related routines eventually finish processing and call `wakeup()` to mark the processes that are waiting for that event as runnable. Internal data objects are often created during processing. These objects are passed from routine to routine, and are eventually stored in a queue or a buffer so that the process can receive the contents of the objects.

We use the address for each internal data object as the identifier of an event. Assigning an identifier to each interrupt does not work properly because one interrupt is often accompanied by multiple events. The reception of multiple packets from a network is a typical example of such cases. Internal data objects are always created for events even in these cases. Thus their addresses can be used to track asynchronous event processing.

We have added a global variable named `svdevent` (stands for “served event”) that shows an event identifier. We have embedded codes in all places where new internal data objects are created so that the addresses of objects are recorded and that a profile data buffer is allocated and associated with each object. We have also added codes to places where internal data objects are discarded. These codes maintain a value of `svdevent`. We have also modified the interrupt handlers so that the value of `svdevent` can be properly saved and restored upon interrupts.

4.4 Switching profile data buffer

The monitor and sampler switch the profile data buffer based on the values of `svdproc` and `svdevent`. Table 4.1 summarizes switching of the profile data buffer.

The value of `svdproc` precedes that of `svdevent`. If `svdproc` shows that <process> needs to be profiled, the monitor and sampler store profile data in the profile buffer for that process. `Svdproc` is OTHER when a kernel is serving processes not being profiled; the monitor and sampler aggregate profile data to a special profile buffer for those processes. When both `svdproc` and `svdevent` are NULL, the kernel is serving system-wide jobs such as scheduling; the monitor and sampler store profile data to a profile

<code>svdproc</code>	<code>svdevent</code>	profile buffer
NULL	NULL	Kernel
NULL	<event>	Asynchronous event
<process>	NULL	Process
<process>	<event>	Process
OTHER	NULL	Other processes
OTHER	<event>	Other processes

Table 4.1: Switching of profile data buffer.

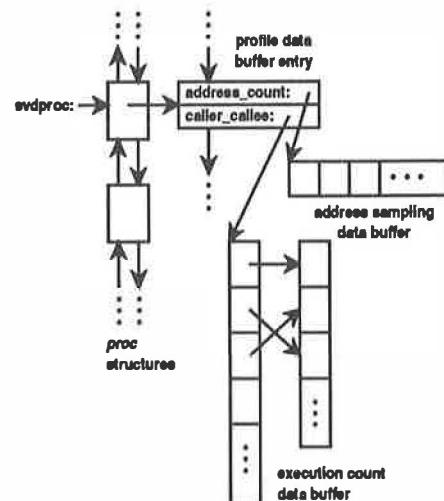


Figure 4.3: Data structures of profile data buffers for processes.

data buffer for the kernel itself.

4.5 Structure of profile data buffer

Because a process usually runs over a long period of time, we designed the profile data buffer that is allocated for each process to be space efficient. The data structures for asynchronous events, on the other hand, are designed to be time efficient. This is because each buffer is expected to be short-lived and the buffer contents need to be merged quickly into the profile data buffer for a process.

Figure 4.3 shows how we organized data structures to store the profile data for each process. PKPROF associates each buffer entry with each process through the `proc` structure to provide monitor and sampler access to the buffer entries. `Svdproc` points to a `proc` structure showing the currently served process. Each `proc` structure contains a pointer to the buffer entry for that process. The monitor and the sampler refer to the entry using that particular pointer. The entries themselves construct a singly-linked list so that each entry can be accessed after the termination of the process.

Each buffer entry contains two pointers: a pointer for the address sampling log buffer and a pointer for the execution count log buffer. The structure of both log buffers are identical to those of KPROF and are used to record gprof compatible profile data.

Figure 4.4 shows the data structures storing the profile data for asynchronous event processing. After this, we will refer to each profile data buffer for an asynchronous event as PDB.

Internal data objects, such as the `mbuf` structure, and PDBs are associated in the hash table. Each PDB is composed of a list of the statistics array. Statistical

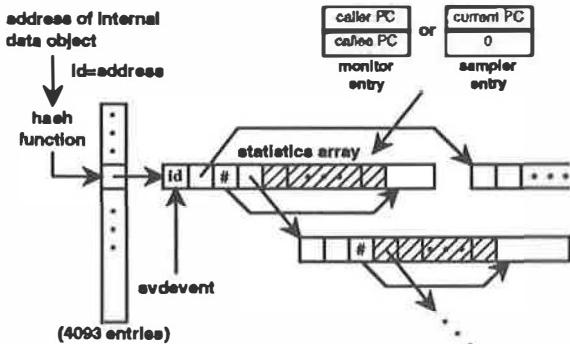


Figure 4.4: Data structures of profile data buffers for asynchronous events.

entries are stored into the array sequentially. The entry can hold a caller-current PC (program counter) pair for the monitor or current PC for the sampler. The sequential structure of the array enables the profiler routines to access contents quickly. Note that svdevent actually points out a PDB so that the monitor and sampler can quickly store the profile data in the PDB without looking up the hash table.

The internal data objects are collected and assembled into other objects for some event processing, such as network packet reception. In these cases, we also collect PDBs associated with all the collected objects and aggregate the contents of the PDBs into a PDB associated with the newly created object. The data structures make aggregation efficient since it only needs to link PDBs by pointers.

4.6 Asynchronous event attribution

A process is found to benefit from each asynchronous event when it uses the output of asynchronous event processing. An asynchronous event handler stores internal data objects, e.g. *mbuf* structures, in an appropriate place, e.g. a queue for a socket, and calls *wakeup()* (See Figure 4.2). When a process removes the internal data objects from places, PKPROF reads the contents of PDBs attached to objects and merges them into a profile data buffer for that process. All the resources consumed for internal data objects are consequently attributed to that process.

4.7 Profile utilities

PKPROF provides two utility programs named *pkgmon* and *pkgprof*. *Pkgmon* is used to conduct profiling. *Pkgprof* is a profile report generator.

We modified the *kgmon* program to implement *pkgmon*. The operating system was originally equipped with *kgmon* to control KPROF using the *sysctl()* library routine, offering miscellaneous services to control kernel behavior³. The controlled

Service Name	Description of parameters
GPROF_ADDPID	Add a process to be profiled
GPROF_PIDMAP	Array of PIDs which have been measured
GPROF_COUNT2 .<PID>	Array of statistical program counter counts of specified process.
GPROF_FROMS2 .<PID>	Array indexed by program counter of call-from points of specified process.
GPROF_TOS2 .<PID>	Array of struct tostruct describing destination of calls and their counts of specified process.
GPROF_WAIT	Wait until: <ul style="list-style-type: none"> • specified time is elapsed • specified processes exit • all process being profiled exit.

<PID>: Process identifier

Table 4.2: Added *sysctl()* commands.

```

pkgmon -x [options] cmd arg ...
      Execute cmd with arg and profile it.
pkgmon -p [options] pid ...
      Specify processes to be profiled.
pkgmon -a [options]
      Profile all processes.
pkgmon -e [options] path
      Profile all processes specified by path.
pkgmon -D [options]
      Dump contents of profile data buffers in a kernel to files.

Options:
  -r      Profile forked processes in addition to a
          parent process recursively.
  -h <number> Profile as much as <number> processes.
  -t <second> Profile until <second> time elapses.
  
```

Figure 4.5: Command syntax of *pkgmon*.

functions include resetting KPROF, starting it, stopping it, and reading the profiled data. We have added new services to the *sysctl()* library routine (and the *_sysctl* system call) and modified the *kgmon* command program so that a user can control PKPROF-specific functionality such as specifying the processes to be tracked and reading process-labeled profile data. Table 4.2 lists the services that have been added to the *sysctl()* library routine. Figure 4.5 summarizes the command syntax for *pkgmon*.

Pkgprof is a front-end processor of *gprof*. We have made no modification to *gprof*. *Pkgprof* collects profile data files and passes them to *gprof*. It supports seamless profiling by merging the user-level and kernel-level profile data of a process into one profile data file (in *gmon.out* format) and passing it to *gprof*. Figure 4.6 summarizes *pkgprof* command syntax and functionality.

4.8 Example Measurements

Figure 4.7 shows a typical procedure to measure seamless profiles. The first command (line 1) executes a *cat* program and starts profiling it. *Pkgmon* reports the process identifier of the executed program (line 2). When the measured program terminates, *pkgmon* also

³The library routine actually issues the *_sysctl* system call to control kernel functionality, but the operating system only documents the *sysctl()* specification.

```

pkgprof [gprof-options] [options] pid...
    Make profile reports for pid.
pkgprof [gprof-options] [options] path...
    Make profile reports for process of which executable is path.
pkgprof [gprof-options] [options] command-name...
    Make reports for process of which name is command-name.

Options:
  -M [/vmunix]      Specify a kernel file
  -d <dir>          Specify a directory where profile data files
                    reside.
  -r                Make profile reports for forked process
                    recursively.

```

Figure 4.6: Command syntax of pkgprof.

```

1: $ pkgmon -x ./bin/cat /x/bigfile >/dev/null
2: 137 ./bin/cat /x/bigfile
3: $ pkgprof 137
4: Created gprof.out.137.cat
5: $ ls
6: gmon.out          gmon.out.137.cat      gmon.out.k.total
7: gmon.out.-k.kernel gmon.out.137k.cat
8: gmon.out.-k.other  gmon.out.137u.cat
9: $ rm *
10: $ ps ax | grep nfsiod
11:   63 ?? Ss    0:03.39 nfsiod 4 (nfsiod)
12:   64 ?? I     0:01.47 nfsiod 4 (nfsiod)
13:   65 ?? I     0:00.73 nfsiod 4 (nfsiod)
14:   66 ?? I     0:00.77 nfsiod 4 (nfsiod)
15: $ pkgmon -t 120 -p 63
16: $ pkgprof 63
17: Created gprof.out.63.nfsiod
18: $ ls
19: gmon.out.-k.kernel gmon.out.63k.nfsiod  gprof.out.63.nfsiod
20: gmon.out.-k.other  gmon.out.63u.nfsiod
21: gmon.out.63.nfsiod gmon.out.k.total
22: $

```

Figure 4.7: Sample run of profile measurement.

terminates. Reports are generated by running `pkgprof` for the `cat` process (line 3).

Lines 6 through 8 show files created by measurement. `Gmon.out.137k.cat` and `gmon.out.137u.cat` are process 137's profile data files for user-level and kernel-level, respectively. `Gmon.out` is another user-level profile data file; its contents are the same as those for `gmon.137u.cat`. `Gmon.out.137.cat` contains information on measurement such as program name and process identifier; `pkgprof` refers the information to process profile data files. `Gmon.out.-k.kernel` is a profile data file for the kernel's system-wide jobs. `Gmon.out.-k.other` is a kernel-level profile data file for not-profiled processes.

Lines 10 through 21 show another example of measurement. A sever process `nfsiod` is profiled in this example. A process 63 has been specified to be profiled for 120 seconds at line 15. Line 19 through 21 show files created by measurement. Note that file `gmon.out` is not generated by this measurement since the profiled `nfsiod` process is still running after measurement.

5. Discussion

5.1 Overhead

Overheads for `pkprof` depend on many factors, such as machine architecture, kernel implementation, sampling rate, and event rate. The goal of this section is to discuss the overheads as generally as possible.

Module	Kernel	KPROF Added to Kernel	PKPROF Added to KPROF
Monitor (<code>_mcount()</code>)	-	145	48
Machine Instruction	-	46411	7611
Total tick			
Sampler (<code>statclock()</code>)	128	25	48
Machine Instruction			

(1tick = 15.625msec)

Table 5.1: Results of static analysis and execution time measurements of monitor and sampler.

We will examine KPROF and PKPROF closely and evaluate the overheads for PKPROF using those of KPROF as standards. Since the implementation of KPROF is fairly general and portable, this evaluation should give the reader clues to estimating the overheads of `pkprof` in her environment.

In the following discussion, measurements have been undertaken on DECstation 5000/200 (MIPS R3000, 25MHz clock) with 16MBytes of main memory and a RZ56 600MByte SCSI disk drive.

Execution time overheads

PKPROF adds 4 kinds of codes to KPROF as follows:

- monitor: codes to refer `svdproc` and `svdevent` and to switch the profile data buffer
- sampler: (same as the above)
- process tracking
- event tracking

Table 5.1 show the machine instruction steps of the monitor and the sampler for both KPROF and PKPROF. It also shows the average execution time of the monitor for each call in tick⁴. The time is measured while a test program reads files and writes them to `/dev/null`⁵. Time measurement results reflect the dynamic environments of the platform, such as cache-miss and bus availability as well as the monitor's typical execution traces. The monitor suffers a 33% text and 16% execution time increase in implementing PKPROF. Note that the increase accounts for 3.1% of the total kernel CPU time. The sampler suffers a 31% increase in text.

Table 5.2 shows the results of static analysis and execution time measurements of process tracking codes. The average execution time for each routine is

⁴The execution time results are measured by using the sampler. We have collected and classified the sample counts for each PC value.

⁵The test load for measurements is as follows:

- Step 1. Read a 256KBytes file and write it to `/dev/null`.
- Step 2. Repeat Step 1 for the same file 15 times.
- Step 3. Repeat Step 1 and 2 sixteen times.

Thus the test program reads 16 different 256 KByte-long data files.

Function	No. of instructions		No. of ticks		called
	Original	Added	Original	Added	
cpu_switch()	95	2	373	0	295048
execve()	460	9	179	3	33605
exit1()	266	9	117	5	33571
fork1()	366	5	134	2	33580
idle()	22	2	59459	5	154445
mi_switch()	135	4	83	10	261477
switch_exit()	22	2	3	0	33571
OTHER	222863	-	153975	-	158127999
total	224229	33	214323	25	158973296

Table 5.2: Results of static analysis and execution time measurements of context switching routines.

Function	No. of instructions		No. of ticks		called
	Original	Added	Original	Added	
interrupt()	120	12	0	0	15119
kn02_intr()	160	4	2	0	15113
lerint()	304	32	14	2	7728
leread()	125	17	1	0	7731
lpinit()	418	98	3	0	7567
udp_input()	467	29	1	0	2577
tcp_input()	1879	13	0	0	14
#receive()	942	54	18	0	2588
pkprof_a_init()	0	43	0	0	-
pkprof_alloc_buf()	0	80	0	0	-
pkprof_attr_seg()	0	205	0	158	-
pkprof_attribute()	0	211	0	8	33683
pkprof_check_halt()	0	47	0	0	-
pkprof_check_idx()	0	161	0	1	2700
pkprof_free_buf()	0	47	0	0	-
pkprof_hash()	0	122	0	4	18120
pkprof_lookup()	0	65	0	3	31293
pkprof_make_total()	0	209	0	0	-
pkprof_reset()	0	99	0	0	1
OTHER	218299	-	6440	-	1297776
Total	222714	1548	6479	176	1442010

Table 5.3: Results of static analysis and execution time measurements of kernel routines for UDP packet receiving.

measured under the same conditions as Table 5.1. We modified 8 kernel routines listed in the table to maintain the svdproc value. These functions suffer a 2.4% text and 0.04% execution time increase in total. Note that the total execution time for these routines accounts for only a small fraction of the total kernel CPU time, that is, about 0.02%. The process tracking codes, therefore, incur a small overhead on system performance.

Asynchronous event tracking codes need to be embedded to each event handling module. The modification heavily depends on specific kernel implementation. Therefore, the overheads for such codes are difficult to evaluate in a general manner. We selected UDP packet receive processing as an example of event handling in the following discussion.

Table 5.3 shows the results of static analysis and execution time measurements of kernel routines related to Ethernet UDP packet receiving. The execution time for each routine is measured in receiving UDP packets of 4 KByte data. The processing of a UDP packet bigger than 1500 bytes that is received via Ethernet needs to have fragments of the packet collected. The measured CPU time overheads, therefore, includes those for concatenating profile data buffers.

Memory requirements

Both KPROF and PKPROF add codes to the kernel. The text size of the kernel with no profiling for our platform is 775 262 bytes. Sizes with KPROF and PKPROF are 803 022 bytes and 816 142 bytes respectively.

PKPROF allocates a profile data buffer for each process (Figure 4.3) when a process is specified to be profiled. It also allocates the same size for two buffers to store the profile data attributed to the kernel itself and the processes not being profiled. The size required for the buffer depends on the address range of the kernel text. PKPROF allocates 1/2 the size of the address range for the execution count data buffer (caller-callee counts) and about 3/4 for the address sampling data buffer.

The address range size of the sample kernel, for example, is 897 048 bytes. Sample implementation first allocates two 1 112 328 byte buffers for the kernel and not-profiled processes. If three processes are profiled, PKPROF allocates a 3 336 984 byte memory for 3 buffers in addition to the first two buffers.

Recording the profile data for each asynchronous event, on the other hand, needs much less memory. Each asynchronous event is generally processed quickly. That is, fewer kernel routines are called and less sampling is made during event processing than during process life. PKPROF, therefore, allocates a small amount of memory for each event. Since such memory is recycled for process-event attribution, the total amount of memory is limited by the throughput of the system's event processing. The maximum memory theoretically needed for profiling asynchronous events depends on the number of internal data objects that a kernel can allocate simultaneously, the number of functions that participate in each event processing, and the time that each event processing takes.

We again selected UDP packet processing as an example to be able to discuss the PKPROF memory requirements more concretely. PKPROF allocates a statistics array (Figure 4.4) for each *mbuf* chain. Each statistics array is 4096 bytes long and can hold 1020 statistical entries. The actual number of statistical entries used in storing the profile data depends on both the implementation of UDP packet processing and the sampling rate.

Measurements on our platform have shown that, for 4 KByte-data UDP packets received at a maximum processing rate (149 packets/s, See "Overall performance degradation" section), the maximum (average) number of caller-callee entries and current PC entries per *mbuf* list are 533 (94) and 2 (0.2), respectively. The 1020 entries are sufficient to hold these entries. While receiving, on average 127 *mbufs* and 333 PDBs simultaneously exist in the kernel

File size (bytes)	Read throughput (Mbytes/s)					
	Normal kernel		Kernel with KPROF		Kernel with PKPROF	
	Disabled	Enabled	Disabled	Enabled	Disabled	Enabled
256K	0.80	0.80	0.78	0.79	0.78	0.78
1M	1.06	1.05	1.04	1.05	1.04	1.04
4M	1.21	1.21	1.21	1.21	1.19	1.19

Table 5.4: Performance impact on file read throughput.

space. These PDBs construct 106 PDB-lists on average, and they accounts for 1 363 968 bytes in total.

Overall performance degradation

Table 5.4 shows the performance impact of PKPROF on file read throughput. It also shows KPROF's impact for comparison. We measured read throughputs of files on a disk for various sizes. We arranged the measurements carefully so that the buffer cache never hit. The results show that both KPROF and PKPROF suffer less than 3% decrease on file read performance.

Table 5.5 shows the performance impact of both PKPROF and KPROF on UDP packet receipt throughput. We used two communication processes for measurement; a receiver running on the platform received UDP packets infinitely and a sender running on another computer sent UDP packets to the receiver via the Ethernet. We determined the maximum throughputs by changing packet transmit rate until the receiver began dropping the packets. The results show that KPROF suffers about a 60% decrease on 64 byte-data UDP packet receive performance and PKPROF about 80%⁶.

5.2 Server process

Server processes that work for other user programs instead of the kernel are common on many of today's operating systems. PKPROF is able to show to what extent kernel resources are consumed by such servers, but cannot tell for what process the servers have worked.

Seamless profiling can be extended to deal with such cases. The kernel places a mark for every data packet passed via interprocess communication. When a server process is scheduled, the scheduler knows which data packet has triggered the server. It can identify for which user program the server is working by referring to the mark. As this method may cause degradation in server performance, markers (e.g. address of passed data) and reference methods should be carefully selected.

⁶When disabled, PKPROF shows less performance impact than KPROF on 64 byte-data UDP packet receiving. We have no idea of the reason.

Packet data size (bytes)	UDP packet read throughput (packets/s)					
	Normal kernel		Kernel with KPROF		Kernel with PKPROF	
	Disabled	Enabled	Disabled	Enabled	Disabled	Enabled
64	3270	2080	1320	2240	670	670
4096	292	292	252	292	149	149

Table 5.5: Performance impact on UDP packet receive throughput.

5.3 Call-graph support

PKPROF reports on seamless profiling results in the same format as gprof. Current implementation can report on call-graph profiles across the user-kernel boundary passed through via system calls. We have modified the monitor codes of the system call trap handler so that they are able to store the addresses of the caller (in user-space) and the callee (i.e. the trap handler) in the profile data buffer properly. The gprof report generator constructs a call-graph across the boundary based on caller-callee information. The call-graph for other services such as page faults needs modification to the interrupt().

5.4 Related tools

Many tools, such as sar of System V and various xxstat programs of BSD, that report kernel statistics have been used to monitor and tune systems [6]. These tools are also useful to measure the kernel resource consumption of a program if it were possible to control the system environment precisely.

If a target operating system kernel clearly organizes process switching and accesses internal data objects in a consistent manner, pkprof can be implemented in a mechanical way. For example, 4.4BSD-Lite uses mbuf structures for network handling in a consistent fashion; similar code patterns appear in many places. In such cases, modification to the kernel to allow pkprof to be implemented also results in embedding the same pattern of codes into many places. ATOM[7] might be useful in such cases.

6. Conclusion

We have proposed a new profiling facility called process-labeled kernel profiling (pkprof). Pkprof measures which resources are used in an operating system kernel for which user program. Seamless profiling, one of pkprof's applications, measures the resource consumption of a program both in user space and kernel space. A programmer is able to understand the impact of a program on kernel CPU usage from the output of seamless profiling. Applying pkprof to a group of processes is useful in understanding system behavior and in improving its overall performance.

Implementation of pkprof heavily depends on the structures of a target operating system. It might be

difficult to add a pkprof facility to an existing operating system. We believe that it is necessary for operating system developers to support pkprof in their operating systems and to consider its implementation during the early stages of their development.

Appendix Some details on asynchronous event profiling

PKPROF functions

Table A.1 summarizes PKPROF functions used for profiling asynchronous event processing. When an interrupt handler allocates a new internal data object, it calls `pkprof_hash()` to attach a PDB to the object. When the handler allocates a new internal data object and discard the old one, it calls `pkprof_pass()` to detach the PDB from the old object and to attach the PDB to the new object. When the handler concatenates two internal data objects, it calls `pkprof_merge()` to merge contents of the PDB of one object to the PDB of the other object. If a process is found to benefit from an internal data object, `pkprof_attribute()` is called to merge contents of the PDB of the object to the profile data buffer for that process. If the process is not being profiled, the contents of the PDB are merged into the profile data buffer for "OTHER" processes.

Profiling disk read

Figure A.1 shows the call flow for the kernel routines of a SCSI disk read operation. The process issues a `read()` system call. The kernel routine named `read()` handles this request and starts setting up a disk read data transfer operation. `Bread()` allocates a `buf` structure and a data buffer for the `buf` structure. `Biowait()` calls `tsleep()` on the `buf` structure to wait for data to become available. A SCSI interface eventually interrupts normal execution after DMA transfer from a disk to the DMA buffer. `Interrupt()` is then called. `Interrupt()` saves the last `svdevent`

```

struct PDB *
pkprof_hash(void *)
    Allocate a PDB and associate it with the argument by storing the PDB
    into the hash table using the argument as a hash key. Return a pointer
    to the PDB.

struct PDB *
pkprof_lookup(void *)
    If the argument is associated with a PDB, return a pointer to the PDB.

struct PDB *
pkprof_pass(void *old, void *new)
    De-associate a PDB with old and re-associate the PDB with new.

struct PDB *
pkprof_merge(void *old, void *new)
    Merge contents of a PDB associated with old into
    a PDB associated with new.

void
pkprof_attribute(void *, struct proc *)
    Merge contents of a PDB associated with an internal data object into a
    profile data buffer for a process.
  
```

Table A.1: PKPROF functions used for profiling asynchronous event.

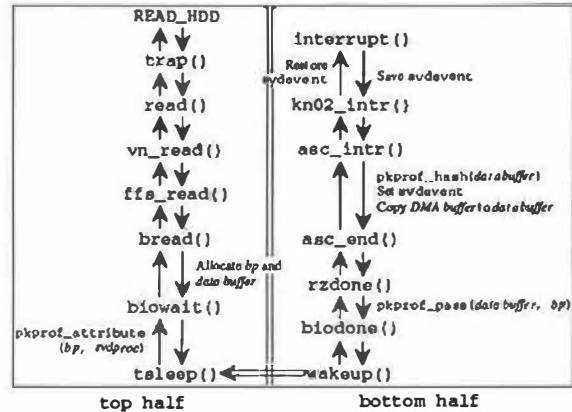


Figure A.1: Call flow of the kernel routines for SCSI disk read operation.

value and sets it to NULL.

`Asc_intr()` is then called to copy data in the DMA buffer to the data buffer passed from `bread()`. `Asc_intr()` allocates and associates a PDB with the data buffer, sets `svdevent`, and starts copying data from the DMA buffer to the data buffer. `Rzdome()` detaches the PDB from the data buffer and re-attaches it to the `buf` structure. `Biodone()` then wakes up processes sleeping on the `buf` structure.

When a process sleeping on the `buf` structure is resumed, the contents of a PDB associated with the `buf` structure are merged into a profile data buffer for that process. The resources consumed in processing the read operation for that `buf` structure are attributed to the process in this way.

Profiling UDP packet receiving

Figure A.2 shows the call flow for the kernel routines of UDP packet receipt operation. The process issues a `recvfrom` system call. The kernel routine named `recvfrom()` handles the system call. `Tsleep()` is finally called from `sbswait()` to wait for a packet arrival. When Ethernet packets arrive, a network interface interrupts normal execution and `interrupt()` is called. `Interrupt()` saves the last `svdevent` value and sets it to NULL.

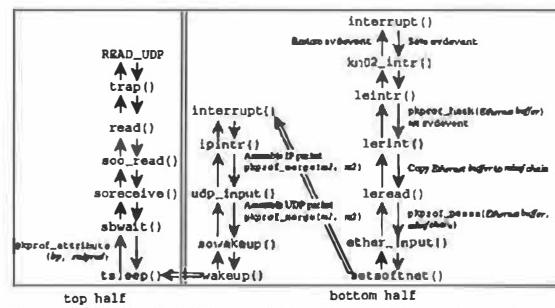


Figure A.2: Call flow of the kernel routines for UDP packet receive operation.

Device driver routines (`leintr()`, `lerint()`, `leread()`, `leget()`) copy the received Ethernet packets from regions of the network interface buffer to *mbuf* chains. `Lerint()` first attaches a profile data buffer to a region of the network interface buffer and sets `svdevent`. That is, an event identifier for a packet is the address of the region. `Leread()` then copies the contents of the region to an *mbuf* chain by calling `leget()`. After copying, `leread()` detaches the profile data buffer from the region and re-attaches it to the *mbuf* chain. `Ethernet_input()` places the *mbuf* chain in the IP packet input queue and posts a software interrupt. This sequence is repeated until all Ethernet packets are processed. Then `interrupt()` restores the last value of `svdevent` and resumes normal execution.

When software interrupt is acknowledged, the network and transport layer routines (`ipintr()`, `udpinput()`, etc.) process *mbuf* chains to assemble, in this case, a UDP packet. `Ipintr()` sets `svdevent` when it removes an *mbuf* chain from the IP packet input queue. When `ipintr()` concatenates two *mbuf* chains, it merges the contents of the profile data buffer for the second *mbuf* chain into those for the first *mbuf* chain. This merging actually chains both the profile data buffers because of their structures (See Figure 4.4). Assembled packets are put into the receive queue of a socket and processes sleeping on that queue are woken up.

When the process waiting for the packet is resumed, it removes the packet from the receive queue of the socket. `Soreceive()` merges the contents of the profile data buffer attached to the packet into the profile data buffer of the running process pointed out by `svdproc`. In this way, resource consumption caused by processing of the packet is attributed to the process receiving the packet.

References

- [1] S. L. Graham, P. B. Kessler and M. K. McKusick, "An Execution Profiler for Modular Programs", Software – Practice and Experience, Vol. 13, 671-685, (1983).
- [2] S. L. Graham, P. B. Kessler and M. K. McKusick, "gprof: a Call Graph Execution Profiler", ACM SIGPLAN Notices 17(6), 120-126, (1982).
- [3] R. J. Hall and A. J. Goldberg, "Call Path Profiling of Monotonic Program Resources in UNIX", 1993 Summer USENIX Technical Conference Proceedings, 1-13, (1993).
- [4] M. K. McKusick, "Using gprof to Tune the 4.2BSD Kernel (May 21, 1984)", Distributed with 4.4BSD UNIX (1993).
- [5] M. K. McKusick, "Measuring and Improving the Performance of Berkeley UNIX (DRAFT April 17, 1991)", Distributed with 4.4BSD UNIX, (1993).
- [6] M. Loukides, *System Performance Tuning*, O'Reilly & Associates, Inc., (1990).
- [7] A. Eustace and A. Srivastava, "ATOM A Flexible Interface for Building High Performance Program Analysis Tools", 1995 USENIX Technical Conference Proceedings, 303-314, (1995).

Author Information

Shingo Nishioka is a research scientist at the Advanced Research Laboratory, Hitachi, Ltd. His interests include programming languages and operating systems. He received his B.S., M.S., and Ph.D. degrees from Osaka University. He can be reached at nis@harl.hitachi.co.jp.

Atsuo Kawaguchi is a research scientist at the Advanced Research Laboratory, Hitachi, Ltd. His interests include file systems, memory management system, and microprocessor design. He earned his B.S., M.S., and Ph.D. degrees at Osaka University. He can be reached at atsuo@harl.hitachi.co.jp.

Hiroshi Motoda has been with Hitachi since 1967 and is currently a senior chief research scientist at the Advanced Research Laboratory and heads the AI group. His current research includes machine learning, knowledge acquisition, visual reasoning, information filtering, intelligent user interfaces, and AI-oriented computer architectures. He holds B.S., M.S., and Ph.D. degrees from the University of Tokyo. He was on the board of trustees of the Japan Society of Software Science and Technology and on the board of the Japanese Society for Artificial Intelligence. He was also on the editorial board of Knowledge Acquisition and on the editorial board of IEEE Expert. He can be reached at motoda@harl.hitachi.co.jp.

Alternately, all authors can be contacted via mail:
Advanced Research Laboratory, Hitachi, Ltd.
Hatoyama, Saitama, 350-03 Japan.

Availability

PKPROF and related documents are available at <http://www.harl.hitachi.co.jp/~nis>.

Cut-and-Paste file-systems: integrating simulators and file-systems

Peter Bosch, Sape J. Mullender

Faculty of Computer Science/SPA, University of Twente, Netherlands

peterb@cs.utwente.nl, sape@cs.utwente.nl

Abstract

We have implemented an integrated and configurable file system called the Pegasus file-system (PFS) and a trace-driven file-system simulator called Patsy. Patsy is used for off-line analysis of file-system algorithms, PFS is used for on-line file-system data storage. Algorithms are first analyzed in Patsy and when we are satisfied with the performance results, migrated into PFS for on-line usage. Since Patsy and PFS are derived from a common *cut-and-paste* file-system framework, this migration proceeds smoothly.

We have found this integration quite useful: algorithm bottlenecks have been found through Patsy that could have led to performance degradations in PFS. Off-line simulators are simpler to analyze compared to on-line file-systems because a work load can repeatedly be replayed on the same off-line simulator. This is almost impossible in on-line file-systems since it is hard to provide similar conditions for each experiment run. Since simulator and file-system are integrated (hence, use the same code), experiment results from the simulator have relevance in the real system.

This paper describes the cut-and-paste framework, the *instantiation* of the framework to PFS and Patsy and finally, some of the experiments we conducted in Patsy.

1 Introduction

Building a fast file-system or introducing new algorithms to an existing file-system is often a difficult task. The designer does not know in full detail what a proposed algorithm does to the overall file-system performance. Normally, the only way to test a new algorithm is to introduce it to an on-line file-system and

This work is supported by the PEGASUS project (ESPRIT BRA 6586) and the BROADCAST project (ESPRIT BRA 6360).

measure the performance effects while the system is in use. If a new algorithm does not work, file-system service may be interrupted.

A better way to test algorithms is to analyze the performance of those algorithms in an off-line simulator. When the algorithm works as expected, it can be integrated in a production file-system. However, the disadvantage of an off-line simulator is that it is hard to build a representative simulator. Usually, simulators are approximations of real systems [21], and results from such a simulator may not show actual system performance.

Recent developments [20, 26, 27, 3, 8] in file-system research have shown us that improving file-system performance is still a hot topic. Many groups find new storage algorithms and report better performance numbers. These performance numbers usually show the effectiveness of those algorithms within the environment they were developed for.

Ideally, to validate those reported performance numbers, other researchers would re-use (parts of) the presented system, insert it into their own environments, and re-evaluate the published performance. In reality, this is well nigh impossible [5]. Often the system code heavily depends on the environment for which it was developed. Through some effort it is possible to port the source code, but then it is hard to set up a similar test environment [9, 29].

To solve both problems, we have designed and implemented an *extensible reference file-system component library* from which we instantiate on-line file-systems and off-line file-system simulators. On-line systems are systems that are in use in real systems, off-line systems are not in use by clients and only run in a controlled environment¹. The component library provides basic components that are required to build a full file-system and the same components are used to build file system simulators. *Helper* components are added to complete an instantiation to a real file-system or simulator. Helper components in a real

¹We have not yet experimented with off-line file-systems.

system deal with actual data movement, while in the simulator they compensate for the lack of “real” data. We have made the component library such that it is easy to extend the library with new algorithms and policies.

The symbiosis of the simulator and the “real thing” helps us to:

- be more confident that simulated off-line performance numbers show real and representative on-line file-system performance numbers;
- easily detect performance bottlenecks of real file-system algorithms by running the same system off-line in an *equivalent* file-system simulator;
- analyze new file-system algorithms off-line before they are integrated into a production file-system;
- be more confident that no side effects are introduced when a simulated algorithm is moved into a real system;
- construct a reference file system: other file system algorithms can easily be integrated and compared to other algorithms in our environment;
- easily migrate algorithms from off-line simulators in real systems: the algorithm does not have to be re-implemented for the real system once implemented for a simulator.

Our initial goal for this work was to build a production file-system and a separate file-system simulator. The production file-system was to be used for ordinary data storage with functionality to store continuous media files, the simulator was to be used for analyzing storage algorithms.

In the original simulator, we analyzed cache-flush algorithms. For that work we replaced the Unix 30-second-update timer policy by a flush policy that keeps dirty data in the cache much longer [4], so-called *write-saving* policies. Unix file-system write traffic is characterized by a high overwrite factor in the first part of a file’s lifetime [2, 10, 17, 22]. Keeping dirty data longer in memory without writing dirty data to disk increases the probability that a block is overwritten through *truncate* and *delete* calls in memory rather than on disk. As a result fewer data blocks are written to disk.

The work showed that replacing the 30-second-update timer by a write-saving policy greatly reduces file-system read latencies. The work claims that disk I/O queues are the main cause of relatively high file-system latencies. Basically, the goal was to get *writes out of the way of reads* simply by writing less data to disk.

The initial simulator used for the experiments used an approximation of a real file system and it used a simple disk model. As is shown by Ruemmler et al. [21], a simple disk model in a simulator may not show the actual performance: the results can be completely useless. Ruemmler et al. reported differences of up to 112% between real and simulated performance. It was obvious we could not trust the earlier analysis.

To present more accurate simulated performance numbers, we decided to build a file-system simulator that simulates a file-system in all details, including a disk sub-system back-end much like HP Pantheon disk simulator [31] and Dartmouth’s disk simulator [13]. We continuously refined the simulator and eventually we ended up with a full file-system. It turned out that this file system shared lots of data structures and algorithms with our real file system, it only lacked data manipulation code: i.e. the simulator was not an approximation anymore. We repeated the write-saving experiments in this version of the simulator and analyzed the performance numbers again.

We then realized that it is important that a system and its simulator are closely related. When policies and algorithms are different, we cannot be reasonably sure that simulated performance shows real performance. Also, when policies and algorithms are analyzed in a simulator, eventually they will have to be migrated into a real file system. If a real file-system uses completely different data structures or is constructed differently, this migration process usually results in a complete rewrite of the policy or algorithm, which may introduce unwanted side effects or new bottlenecks in a real system. When simulator and file-system are derived from a common framework and use similar, if not equal, data and component structures, migrating code becomes a trivial matter.

We now use Patsy, PFS and the cut-and-paste component library as a new way of doing file system development. Algorithms and file system extensions are first analyzed off-line through pre-recorded file-system traces or hand crafted work loads before they are integrated in a production file system. We learn all the effects of algorithms before they are used.

In fact, we use the framework as a reference system and starting point for further work. Quick off-line file-system experiments are performed before we decide to use the algorithm in a production version of the system. For example, in the framework we are currently performing continuous-media storage experiments, and we plan to use the framework for tertiary storage experiments. We feel that having the frame-

work available gives us a head-start for file-system developments: components are already in-place or are available in the library. Components that are written for experiments are always added to the component library for possible later re-use in other experiments.

Currently, we know of one other system that has integrated a file system simulator and a real system. Thekkath et al. [30] describe a file-system simulator that is able to run an existing kernel file-system in a discrete event simulator. Our work is similar to theirs, but we arrived at the same point through a different route. Thekkath's goal was to lift a kernel file-system into a user level simulator and measure the performance of such a system. We started by building a simulator for some file-system experiments, and later integrated the simulator and a real file-system. In our system, we make file-system *development and measurements* easy, whereas Thekkath's system makes *measurements* of existing file-systems easy. The major advantage of Thekkath's system is that they have measured a production file system (Unix FFS [14]) and calibrated their simulator through a production file-system. We are still using an experimental file-system. A second advantage of Thekkath's system is the availability of a real file-system snapshot before an experiment starts. We are upgrading PFS from an experimental file system to a production file-system and we will use snapshots of PFS in Patsy experiments.

The remainder of this paper is organized as follows. Section 2 describes the common cut-and-paste framework. Section 3 describes the added modules to make up a PFS, and Section 4 describes those components added to make up a file-system simulator. Section 5 we describe a simulator that we built through the frame work, we describe some of the experiments we conducted in this simulator and it describes the lessons we have learned when we were building the system. Section 6 summarizes this paper.

2 Cut-and-Paste

Our goal for the component library is to create a reference file system framework from which we can instantiate many possible file system and simulator configurations. This reference file-system framework, therefore, must be easy to maintain and easy to extend. The cut-and-paste framework should not enforce designers to early algorithms and policy decisions.

To build an *open* component library, we decided to implement the system in an object-oriented language. We have created a set of base components that are used in all file-systems and implement default behav-

ior. Derived components implement specific behavior and are accessed through an inheritance chain. We do not allow derived components to enrich the interface of its base component without adding an interface to its base component. This ensures that interfaces remain clean and without cross dependencies to other than the base component dependencies. By restricting ourselves to such a scheme, our system remains upward compatible.

The cut-and-paste component library is best visualized as a collection of objects that are combined into a real or simulated system whenever they are needed. All components are written in C++, are instantiated from their classes and bound to global variables when a system starts. Base components are C++ base classes, derived components are C++ derived classes. The system currently consists of $\approx 55K$ lines of C++ code, divided into 80 classes.

Currently, the following *core* components exist: a *thread scheduler* that allows multiple independent processes in the real or simulated system, a *cache* that implements a full file-system block cache, a *file-system storage* component that defines the storage layout on disk, a *client interface* that defines an abstract front-end to the system, an *abstract file* that "knows" everything about a file when it is loaded into the cache and a *device-driver* that communicates with the simulated or real hardware to read and write data from disk.

For all of the *core* components we have implemented the default behaviour and one or two other algorithms. The core components themselves implement a true *default* file-system. As we are still adding functionality to our system, we expect the number of available algorithms to grow rapidly.

Figure 1 shows most of the core components of the combined file system and simulator. The figure is used as a reference figure: not all components are immediately explained. In particular, all components that are not located inside boxes labeled Patsy or PFS are common to both simulator and system and are part of the cut-and-paste framework. The cut-and-paste components are explained in the remainder of this Section. Boxes labeled PFS are explained in Section 3 and boxes labeled Patsy are explained in Section 4. Since interfaces between objects can be quite large we decided not to describe them here and we refer to the source distribution.

Thread scheduler

The thread scheduler implements threads, synchronization primitives and real or virtual time. Independent file-system processes are given a separate thread of control inside the system. These threads are able

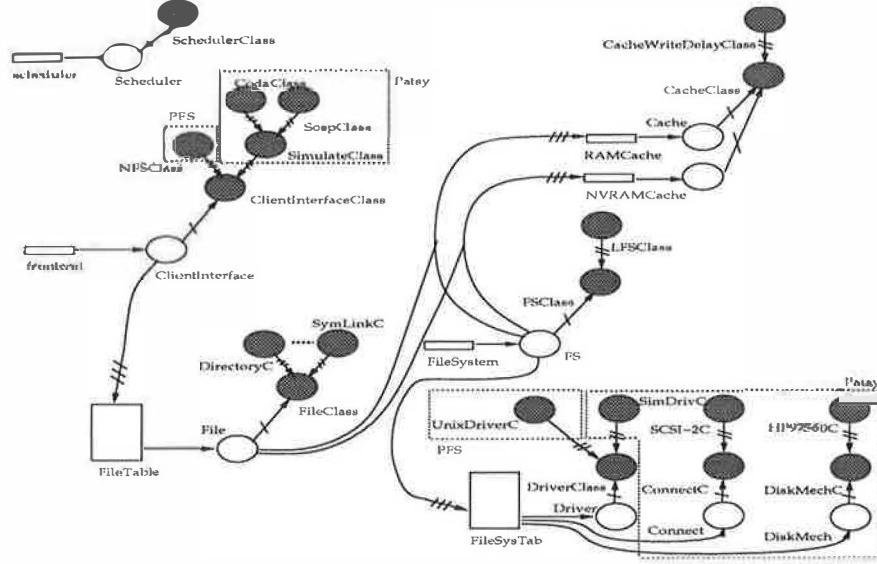


Figure 1: Some of the framework components. Class definitions are shown as shaded ovals, instantiated objects as clear ovals, and global variables and hash tables as clear boxes. Unstriped arrows are plain pointers, single-striped arrows show the relation between the instantiated object and its class definition, double-striped arrows refer to the inheritance chain between the class definitions and triple-striped arrows show which objects call methods in other objects.

to communicate with each other through synchronization primitives: one thread can make others runnable. Threads can also suspend by yielding the processor for some amount of time. Finally, the scheduler knows what the current time is for a real system and it defines virtual time for a simulator.

The synchronization primitives are based on events. Each thread can pick a unique event and block on it. Once a thread has blocked itself, another thread *signals* the event through the scheduler to make the thread runnable again.

Internally, the scheduler maintains a queue of all delayed threads and makes them runnable whenever the timers expire. If the scheduler is configured in a simulator, thread-timers only expire when there are no other threads to run: virtual time is increased to the timer-expire time of the first thread on the delayed queue. When the scheduler is configured in a real system, timers expire in real-time².

External events are also managed by the scheduler when it is configured in a real system. In Unix, a file-descriptor is associated with a thread and whenever a message arrives on the file-descriptor, the thread associated with the file-descriptor is made runnable.

Currently, the scheduler provides random scheduling. It picks a random thread from the runnable set, whenever it has to schedule the next thread. When

other scheduling policies are required (e.g. when files with real-time constraints are introduced), a derived scheduler class can implement a different scheduling policy.

Caches

The cache modules are used to administer and maintain a file-system block cache. It provides interfaces to administer all dirty, non-dirty and free blocks in lists, and it provides interfaces to allocate blocks from the cache. Also, when blocks are allocated from a full cache, it decides which blocks are replaced and flushed.

The base cache component implements LRU lists to maintain all dirty and non-dirty blocks. Blocks are first allocated from the non-dirty list, and when there are no non-dirty blocks available, the cache initiates a cache flush through the oldest dirty block. To flush, it calls an internal method that may be overloaded by different policies. For example, we have implemented a flush policy that flushes the whole file rather than a single block when a block flush is initiated.

Specific persistency requirements can be implemented in derived components that call into the base component to initiate cache flushes. For example, the Unix SVR4 30-second-update timer policy is implemented through a derived class that examines the contents of the cache every couple of seconds. When

²Threads are made runnable in real-time.

it detects that there exists a dirty block older than 30 seconds, it flushes the file associated to the oldest block.

Different cache administration policies are easily implemented by re-implementing the replacement methods of the base-class in a new derived class. For example, to experiment with different replacement policies (e.g. RR, LFU, SLRU, LRU-K or adaptive [12]), only those functions that deal with LRU replacement need to be replaced from the base-class.

The difference between a simulated cache and a real cache is the lack of a data pointer in the simulated case. In all cases where data is moved between buffers, the simulator delays the current thread for the amount of time it would take (based on the system hardware configuration) to copy the data. In a real system, a large chunk of (physical) memory is allocated and divided over all the cache blocks when the system starts.

Storage-layout

The storage-layout component is responsible for defining a file-system layout on a raw disk. This component knows the actual location(s) of file-system metadata, and is able to store and retrieve information from one or more disks. It is consulted whenever something needs to be done with a raw disk.

The base storage-layout class is only an interface: it does not implement an algorithm. Specific layouts are implemented through derived classes. The interface to a storage-layout class is defined such that for all layout and policy decisions, there exists a virtual method in the base-class.

Currently, we have implemented a segmented Log-structured File-System (LFS) [20, 26]. This system stores file-system updates to the end of the log, and is able to find files through an inode-file (IFILE). The log-cleaner can be replaced and is plugged into the LFS component when the system starts up.

To implement other storage-layouts (such as a Unix FFS [14], EFS [28], or journaling file-systems), a new derived storage-layout class needs to be written that defines a new storage-layout on disk. This derived class needs to implement all of the methods defined in the abstract storage-layout class.

A storage-layout module can also be instantiated for a simulator. In this case, all information that would have been read or written to disk is simulated by making educated guesses. If, for example, a file is accessed that is not yet known by the storage-layout module, it picks a random location on disk. Once an initial location has been chosen for a file, the simulator sticks to those addresses.

Abstract Client Interface

The abstract client interface provides the basic file-system interface. There are functions to *open*, *close*, *read*, *write* or *delete* a file and there are functions to manipulate an hierarchical name-space.

The abstract client interface initiates the loading of a file from disk when it is first accessed. It calls into the file system module to read the file's inode into memory. Once the file is in memory, the component stores a reference to it in a global file table.

Furthermore, when the file has been loaded into memory, the abstract client interface maps all incoming user requests to the memory representative of a file.

Files

Abstract client requests are dispatched to so-called *instantiated* files. An instantiated file is used to control a file that has been loaded into the file-system cache. It may contain a memory copy of the file's inode, references to cached file data, and it contains a set of functions to perform operations on a file, such as a *read*, *write* and *flush* method.

As there are many file types in current file-systems (e.g. ordinary Unix-like files, directories, symbolic links, multi-media files, and administrative files) each with different access patterns and behavior, we have implemented each file type in a separate derived class. All components derive basic file functionality from the base file. This structure allows us to separate policies that optimize access to a particular file type. An example of this is a multi-media file. If ordinary cache policies are used on a multi-media file the whole cache would fill up with this data. A multi-media file prevents this from happening by implementing other cache policies.

Pei Cao *et al.* has shown that two-level file-caching can be beneficial to overall file-system performance [7]. In this work, cache replacement policies are delegated to a user-level manager process. In our approach we have the opportunity to delegate the policy decisions to a particular file - the manager can be implemented inside the file-system. When a file is opened, a client can inform the file-system to use a certain replacement policy. This approach can give us a fine-grain control over cache replacement policies.

When a file is requested by a client, the file-system front-end examines the file type of the requested file (through its inode or other administrative means) and instantiates an object of that type to manage the file while it is in core.

A file is called *active* if the instantiated file spawns a thread of control that works independently inside the

file-system. Such functionality is especially useful if files are manipulated that have timing constraints on them. In a multi-media file, for example, the thread of control may take care of cache pre-loading and can even negotiate with (remote) clients and other modules in the file-system to establish a certain *Quality of Service (QoS)*.

3 Pegasus File-system

The base components in the cut-and-paste library do not make up a complete system: they lack interfaces to the environment. To complete such a system, *helper* components are added to the component library that glue all the components into the environment.

The system glue currently only consists of two parts: the system needs a *real* user interface, a PFS client interface and it requires a real disk-driver to access a real disk.

PFS Client Interface

We use NFS [23] as the external PFS interface. We have constructed a full NFS client interface class, which is a derived class from the abstract client interface class. The NFS class spawns a number of threads that wait for incoming mount and NFS requests. Whenever a request is received, the call is dispatched to one (or more) calls in the abstract client interface. Each thread in the NFS component acts as a representative of a client while the request is in progress.

In the future we will construct a client interface that enables strict consistency. This client interface will use many of the techniques that are already in use by Sprite [16] and Mike's File System (MFS) [6]. By using client caching we hope to reduce the amount of network traffic and file latency. We will realize this within the cut-and-paste framework so that we can simulate client/server interaction and client cache performance.

Disk-driver

Real disks are accessed through disk-drivers. Disk-drivers implement one or more disk queues and send new operations to disks whenever they are ready to service new requests. They can implement disk queue scheduling policies to optimize disk I/O queue time (e.g. SCAN, C-SCAN, LOOK, C-LOOK [11, 25]) or guarantee real-time delivery of data through algorithms such as scan-EDF [18].

Currently, only one disk-driver exists. This driver implements a combined read-write queue and schedules I/O requests through the C-LOOK scheduling policy. It uses a Unix-file (ordinary file, or raw-device) as back-end.

4 Patsy

Patsy is the instantiation of the cut-and-paste library to a file-system simulator combined with some helper components for off-line file-system simulation. In particular, we added components that simulate real disk-drivers and disks, a component that simulates the connection between the host and disk sub-system, and a component hierarchy that is able to read a particular trace file and dispatch it to the simulator. We are simulating only a small subset all hardware types. We hope to integrate our file-system simulator into HP's Pantheon disk-simulator and use that as our disk back-end because Pantheon simulates a wide variety of storage hardware.

Simulated disk-drivers

Simulated disks are accessed through simulation disk-drivers. These disk-drivers provide the same functions as their real counterparts, but also provide mechanisms to simulate the sending and receiving of operations from disk. The simulated disk-drivers have exactly the same interface as a real disk-driver: the differences are in the internal implementation. The system itself does not know it is communicating with a "fake" disk.

Simulation disk drivers package disk operations in I/O-request data structures. The I/O-request data structures contain all the relevant information for the disk simulator to simulate a disk read or write and contain timing information to measure the performance of the I/O operation.

Before an operation is activated on disk, the disk-driver acquires the host/disk connection and simulates the sending of data. This is required as many more entities can make use of the same host/disk connection. If the connection is already in use, the disk driver waits until the connection is released again. Finally, it sends the I/O request to the disk itself and activates it.

The disk will perform the I/O request and later transmit the results to the disk-driver. For this, the disk acquires the host/disk connection and simulates the transmission of the I/O request back to the driver. Finally, the simulated disk-driver resumes the original caller and informs it of the I/O results through the I/O request data structure.

Simulated disks

The disk component in the simulator acts as a representative for a real disk. A simulated disk component knows about heads, tracks, sectors, rotational speed, controller overhead and it may implement disk cache policies.

Internally, a disk is modeled by a separate thread of control that waits for work to arrive from external sources. Whenever a read or write request arrives at the disk, the controller unpacks the request, seeks to the correct cylinder or switches heads. Next, the disk waits for the rotational delay and reads or writes data to disk. Finally, the disk transfers the data to the host, or signals the host that a write has completed.

It is obvious that in the simulated world, no real data is moved to and from disk. We simulate this by delaying the thread of control by the amount of time it would have taken to transfer the data.

Currently, we have implemented an HP97560 disk [21, 13] as a separate disk class. This disk is equipped with a 128KB internal cache that can be used for immediate reported writes (writes that complete once data arrive in the disk's internal cache) and a read-ahead policy (when there are no more outstanding requests, the disk reads the next 4KB following the last read).

Connections

Connections are the links between the host and the disk sub-system. Connections are used to transfer data, requests and responses between disk and hosts. They also arbitrate if there is more than one controller that wants to send data over the same connection to simulate connection contention (e.g. SCSI bus contention).

Again, as no real data can be moved through a connection, the connection delays the thread of control by the amount of time it would have taken to transfer the data through the connection.

We have implemented a SCSI-2 bus [24]. This bus allows multiple hosts/disks to use the same connection, and it allows hosts/disks to disconnect and reconnect during a single SCSI transaction. The bus simulates a bus transfer speed of 10MB/s.

Work loads and traces

File-system traces are collections of records that describe all the activity of a real file-system at some time. These records specify when the operation took place (usually down to the microsecond), and which file-system operation was executed. Usually, file-system traces do not present all the details that are

required for an exact replay. The reason is that, although all parameters can be recorded, doing so would result in prohibitively large trace files: the recording of such trace files would influence ordinary file-system performance too much [15, 17]. When replaying traces, we synthesize those parameters that are missing as best we can (e.g. the initial location of a file on disk, file names, initial layout of the file-system, the exact time a read or write was executed).

We are also considering a component that can be used to hand craft work loads using probabilistic means. This component will, given some inputs, generate a work load and dispatch it to the simulator. The advantage of such a component is that it will improve our confidence in the simulation results.

We have modeled a trace simulation class hierarchy on top of the abstract-client interface. All file-system requests recorded in the file-system traces are mapped to calls in the abstract-client interface. The simulator components currently consist of three parts: a general simulation class that provides performance results gathering and dispatches operations to the abstract-client interface. Furthermore, there are two derived classes: a Sprite class to replay the Sprite traces [2] and a Coda class to replay the Coda traces [15].

Clients are modeled by separate threads of control inside the Sprite and Coda classes. The threads read a part of the trace file, group operations that obviously belong together (such as an *open*, *read*, *read*, *write*, ..., *close* sequence), and call the abstract-client interface to execute the operation on the simulated system. Since all of the trace records have timing information in them, the threads know how long they have to delay themselves before they can dispatch the next operation.

When simulation information is missing (such as the actual time a *read* or *write* operation took place), the client thread makes a guess. In the case of the missing *read* and *write* times, the operations are positioned equidistant between the *open* and *close* operation. We plan to experiment with the position of these operations.

The overall measurements are taken from the general simulation class. This class measures how long it takes before an operation completes. The measurements are shown every 15 minutes of simulation time and of the overall simulation.

Detailed internal measurements are provided by plug-in statistics objects. These plug-in statistics can be activated when the simulator is started and they can provide standard statistics output with or without histograms. Some of the standard detailed statistics objects include histograms of disk queue sizes, cache statistics, and disk rotational delay statistics.

5 Using the component library

We have used the component library for a file-system performance experiments and to construct a real file-system. We built the algorithms for the various experiments in the simulator, and later we instantiated the same algorithm for a real system: we did not have to change anything in the code except for some small additions when data was actually moved.

In this section we describe an experiment we conducted in an instantiated simulator and some of the lessons we learned when we built the component library. The experiments primarily show what kind of simulations are supported. The full results that are presented briefly in this section are subject of another report. We only present the final results of the off-line experiments.

5.1 Delayed write policies

One of the reasons to build a file-system simulator in the first place was to re-do the performance analysis of various delayed write policies, based on ideas described in earlier work [4]. Also, we have conducted the experiments to confirm cache experiment results as reported by Ousterhout [17]. In those experiments, we buffered dirty data longer in the cache with the hope that less data is written to disk. If dirty data stays longer in the cache without being written to disk, changes are higher that file *delete* and *truncate* calls remove the file's dirty data before a flush policy gets a chance to write the data to disk. If less data is written to disk, disk queues become shorter and I/O latencies decrease.

If the fraction of dirty data in the file-system cache increases, read cache hit-rates may be negatively influenced. If there is more dirty data in the cache, non-dirty data is replaced earlier. If overall cache hit-rates drop (i.e. if cache hit-rates are higher for non-dirty caches), more data is read from disk. This leads to longer disk queues and to increased I/O latencies. We analyze this trade-off.

If dirty data remains longer in the cache before being written to disk, more data is lost when the system fails. Earlier work describes how to protect dirty data from a single point of failure in the system through client write caching and by using non-volatile RAM (NVRAM) or a uninterruptable power supply (UPS) in the file-server [4].

We are performing four different experiments with the Sprite traces to analyze the performance effects of these *write-saving* policies. Our first experiment shows the base line performance of an ordinary Unix file-system with a 30-second-update policy. Dirty data is buffered 30 seconds before it is sent to disk

(the write-delay experiment). Next, we equip the file-system with a UPS and only flush a cache block when we are out of non-dirty cache-blocks. This experiment shows the other extreme as blocks are only written when the memory is filled with dirty blocks. Finally, we equip the file-system with 4 MBs of NVRAM and we disallow dirty data to reside in volatile-RAM. If the NVRAM is full and we need free (or non-dirty) blocks in the NVRAM buffer, we flush the oldest dirty block to disk. For the NVRAM case we consider two flush policies: we either flush the whole file associated with the oldest block (with the expectation that on average there is more non-dirty data in the cache), and we flush only the oldest block.

We have included the NVRAM experiment to answer how much file-system latencies improve by using a small NVRAM buffer in the file-server. This question was raised in Baker et al. [1] and left unanswered.

For all experiments, we expected the UPS experiment to perform best. The reason for this is that all of the available cache can be used for write-caching, minimizing the amount of written data. For the other experiments we did not know on beforehand which policy would work best: if too much dirty data is generated, the NVRAM buffer may be the system's bottleneck and reducing the write-delay time when the file-system is busy. In fact, adding NVRAM to the file-system may be counter productive.

We have rebuilt the Sprite file-server as closely as we could [29, 9] through the cut-and-paste component library and we ran the recorded Sprite traces on this version of Patsy. The original machine on which the traces were recorded was a Sun 4/280 equipped with 128MB of main memory, and three SCSI busses that connect to a total of 10 disks. There were a total of 14 file-systems on the set of disks, of which two were clearly hot-spots. For the experiments we have used simulated HP97560 disks and SCSI-2 busses. On all file-systems we ran a segmented LFS. All components used by this simulator are derived from the cut-and-paste framework.

Figures 2–4 show a cumulative distribution of the file-system latencies for the Sprite traces in Patsy (only traces 1a, 1b, and 5 are shown, the others are permutations of these three runs). Each trace represents a 24-hour period of operations and we have measured the time it takes to complete each of the operations. The figures show a cumulative distribution of the latencies. The horizontal bars show the average experienced latency, the vertical bar shows the fraction of operations completed. Figure 5 shows all mean file-system latencies for all the traces.

Each graph in Figures 2–4 have a similar form. All operations that complete within 2-milliseconds

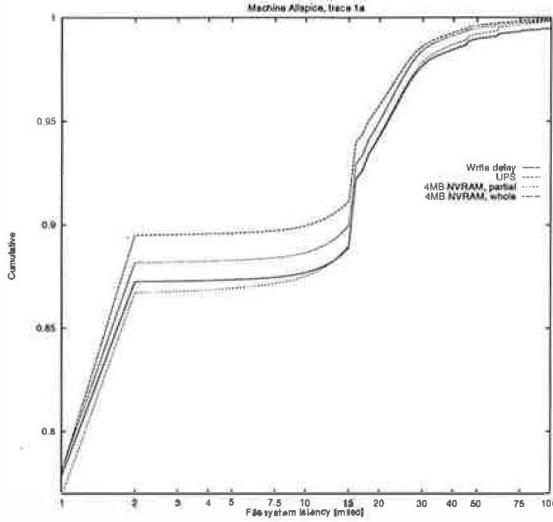


Figure 2: File-system latencies, trace 1a

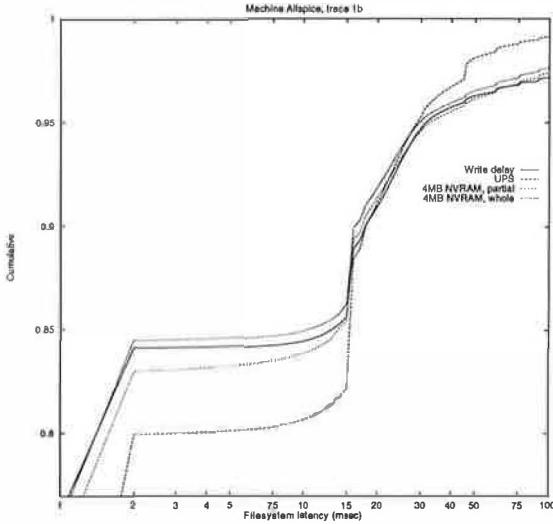


Figure 3: File-system latencies, trace 1b

are serviced from the file-system caches. The 2-milliseconds boundary is the minimal latency when a request is serviced by the disk (SCSI-request decoding). The period up to 17-milliseconds represents the time waiting for the rotation on disk (HP97560 disks spin at 4002 rpm), including the controller overhead. The bump at 17-milliseconds represents the large amount of operations that had to wait for a full disk-rotation. The periods larger than 17-milliseconds are those when the disk queues were longer than one entry or when the disk required head and/or cylinder switches. We are re-evaluating the trace results to present both delays separately.

From the performance numbers shown in Figures 2–4 and Figure 5 we learn that, in general, the UPS

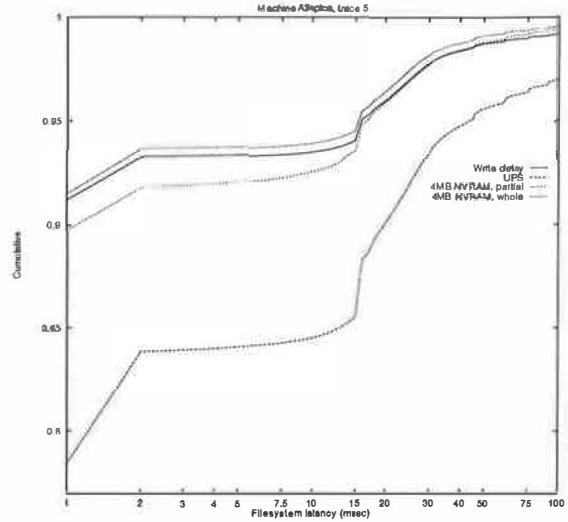


Figure 4: File-system latencies, trace 5

experiment performs better than the NVRAM experiments, which in turn performs better than the ordinary write delay policy. In all but trace 5, a UPS file-system is much faster than the write delay experiment while the NVRAM experiment is only twice as fast. The reason for this is that for most traces disk queues are minimized.

During trace 5, many large *writes* enter the system while there are also a fair amount of *stat* and *read* operations. The write operations fill up all of the available memory (which are not flushed since we are using a naive flush policy) and clutter up memory. This lowers read cache hit rates³, and cause the read operations to stall while data is flushed to disk. This happens to a lesser extent in trace 1b. We found that in general write-saving policies combined with a naive flush policy resulted in lower cache hit rates.

Figure 5 shows that for Trace 1b NVRAM does not help much to reduce file system latencies. In this trace there are many large and parallel write operations. Since dirty data can only be stored in NVRAM, the NVRAM becomes a bottleneck: new writes are waiting for the NVRAM to drain. In some cases we found that the write-back policy deteriorated to a write-through policy.

For the NVRAM case, we measured two different flush policies: whole file (that is: when a file-block is flushed, all dirty blocks of that file are flushed), and partial file (only the selected block is flushed). As expected, whole-file flushes perform better than partial-file flushes as it leaves on average more non-dirty space in the NVRAM buffer. This means that write operations complete sooner as they do not have

³Not shown.

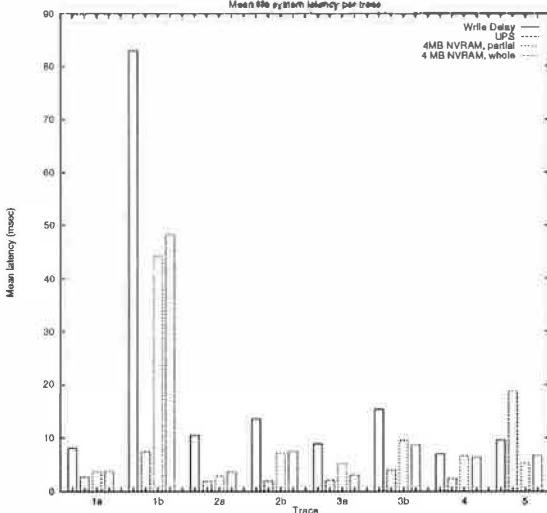


Figure 5: Mean file-system latencies

to wait for data to be flushed to disk first.

In general, delaying write operations reduces disk contention even though there are more cache misses. We showed that because of this, file-system operation latencies decrease. When there is lots of activity in the file-system, there is a possibility that the available memory is cluttered up with dirty data, which increases subsequent file-system latencies as data needs to be written to disk first. To solve this problem, we are experimenting with more aggressive write policies.

5.2 Lessons learned

While building and experimenting with an off-line simulator, we were surprised by a number of performance bottlenecks that would have been hard to find in an on-line file-system. Since simulator and system are one and the same, solving the performance bottleneck in the simulator, also solves the performance bottleneck in the real system.

We ran the Sprite traces on the simulator and we were surprised by the “slowness” of the simulator: it took several hours to simulate one hour of Sprite time. First, we used *prof(1)* to analyze which procedures were accessed most. It turned out that the way we were maintaining the LRU lists was sub-optimal. By carefully analyzing what the system was doing through *gdb(1)*, we detected several short-cuts in list maintenance. This improved simulation time dramatically. To look for this problem in a real system would have been hard. First, it would have been hard to continuously replay the same workload. Second, it would have been hard to run the real system under *gdb(1)* and to analyze step-by-step what the actual

problems are.

A second problem we found had to do with the way a cache was flushed. In our original system, the thread that needed a cache block was also the one that initiated a cache flush and waited for the flush to complete. As more esoteric flush policies were used, the delay for this thread increased. Since we were able to analyze off-line we were able to quickly analyze why some threads were severely delayed. The obvious solution was to make the flush policy an a-synchronous operation. Again, this problem would have been more difficult to find in a real system.

We found the NVRAM contention problem through carefully analyzing and hand-crafting a work load. The other option would have been to allocate a part of a file-server’s memory and simulate the behavior of NVRAM in an on-line system. By being able to simulate behavior off-line, we were able to quickly decide that it is better to equip a file-system with a UPS rather than NVRAM.

The biggest surprise was when we instantiated the component library to a real system: most algorithms ran immediately. This basically means that file-systems can be developed in a controlled environment. The file-system is now in use for real data storage.

It is certainly true that file-system experiments can also be performed in kernel-based file-systems. Current BSD systems can easily be changed to support all kinds of experiments (as noted by an anonymous reviewer). We believe, however, that user-level file-systems are easier to maintain and it is easier to track performance bottlenecks. Developing file-systems in user mode has the advantage that the machine does not halt whenever there is a fatal error. A production version of the system can always be implemented inside the kernel, although we do not expect the performance to improve much. The problem with Unix user-level file-systems, however, is that some parts of the file-system are constructed carefully to hide the absence of threads in most Unix systems. To demonstrate that the current system can also be used as a kernel service, we have inserted the system into the Nemesis micro-kernel [19]. The reason for this integration is to provide Nemesis with a file-system and to perform multi-media storage experiments.

We fully agree that the system described in this paper is a partially an engineering practice (as noted by an anonymous reviewer): we have built a tool to perform file-system experiments. Since related parts of the system are encapsulated in their class hierarchy, changing particular file-system policies is not hard. This gives us an opportunity to quickly learn the effects of new algorithms and file-system configurations (even if we do not have the equipment we are

simulating).

5.3 How to make the results usable

Based on the experiments, we have decided to pursue a real file-system that is equipped with a UPS. We are re-evaluating a protocol to safely distribute dirty data over client and server machine. As is shown in earlier work [4], this allows us to guarantee data persistency in case of a single point of failure, or a global power failure. Evidently, we will re-evaluate this through the simulator.

Once we are happy with the performance results of a more aggressive flush policy, we will migrate the cache policy into PFS and measure the performance relative to a standard 30-second-update policy. In these measurements we will show the performance differences between a simulator and a real system.

We have compared performance differences of system and simulator in a small test environment. The analysis so-far suggests that the results in the simulator have real value and are comparable to real performance. We are working on a full comparison of a simulator and system and we will provide a separate report that validates our approach.

6 Summary and further work

We have presented a file-system reference framework, which we use to experiment with new file-system storage algorithms. The file-system framework can be instantiated to off-line simulators and on-line file-systems. Through off-line simulations, storage algorithms are analyzed thoroughly before being used in a real system. We have constructed the framework such that it is reasonably straightforward to analyze other researcher's algorithms in our simulator.

As an example, we have analyzed several write-saving policies in a simulator, that is instantiated from the framework. We concluded that write-saving policies are beneficial even though cache-hit rates decrease. Currently, we are still analyzing more aggressive write policies before we will put the policies to use in a real system and measure the relative performance of a real system compared to an equivalent simulator.

Acknowledgments

We would like to thank the anonymous reviewers, Tage Stabell-Kulø, Richard Golding, Sjoerd Mullen-dor, Arne Helme, George Neville-Neil, Paul Sijben, Martijn van der Valk, Lars Vognild and the other Pe-

gasi for their support and criticism on earlier versions of this paper.

Availability

The framework is available through the authors. Announcements are made on (pfs-request@pegasus.esprit.ec.org).

References

- [1] Mary Baker, Satoshi Asami, Etienne Deprit, John Ousterhout, and Margo Seltzer. Non-volatile memory for fast, reliable file systems. *Proceedings of Fifth ASPLOS* (Boston, Massachusetts), pages 10–22, 1992.
- [2] Mary G. Baker, John H. Hartman and Michael D. Kupfer, Ken W. Shirriff, and John K. Ousterhout. Measurements of a distributed file system. *Proceedings of the 13th ACM Symposium on Operating Systems Principles* (Pacific Grove CA (USA)), volume 25, number 5 of Operating Systems Review, pages 198–212, October 1991.
- [3] Trevor Blackwell, Jeffrey Harris, and Margo Seltzer. Heuristic cleaning algorithms in log-structured file systems. *Conference Proceedings of the USENIX 1995 Technical Conference on UNIX and Advanced Computing Systems* (New Orleans), pages 277–88. Usenix Association, 16–20 January 1995.
- [4] Peter Bosch. A cache odyssey. M.Sc. thesis, published as Technical Report SPA-94-10. Faculty of Computer Science/SPA, Universiteit Twente, the Netherlands, 23 June 1994.
- [5] Peter Bosch. From ULFS towards PFS. University of Twente, 9 December 1992. A list of notes used to design a new file system.
- [6] Michael Burrows. *Efficient Data Sharing*. PhD thesis, published as Technical report 153. University of Cambridge, DEC 1988.
- [7] Pei Cao, Edward W. Felten, and Kai Li. Implementation and performance of application-controlled file caching. *Proceedings of First Symposium on Operating Systems Design and Impl. (OSDI)* (Monterey, CA), pages 165–77. Usenix Association, 14–17 November 1994.
- [8] Richard Golding, Peter Bosch, Carl Staelin, Tim Sullivan, and John Wilkes. Idleness is not sloth. *Conference Proceedings of the USENIX 1995 Technical Conference on UNIX and Advanced Computing Systems* (New Orleans), pages 201–12. Usenix Association, 16–20 January 1995.
- [9] John H. Hartman. Allspice's configuration, 24 March 1995. Private communication.
- [10] John H. Hartman and John K. Ousterhout. Letter to the editor. *Operating Systems Review*, **27**(1):7–10. Association for Computing Machinery SIGOPS, January 1993.

- [11] David M. Jacobson and John Wilkes. Disk scheduling algorithms based on rotational position. Technical report HPL-CSP-91-7rev1. Hewlett-Packard Company, 16 February 1991.
- [12] Ramakrishna Karedla, J. Spencer Love, and Bradley G. Wherry. Caching Strategies to Improve Disk System Performance. *IEEE Computer*, pages 38–46, March 1994.
- [13] David Kotz, Song Bac Toh, and Sriram Radhakrishnan. A Detailed Simulation Model of the HP 97560 Disk Drive. Technical report PCS-TR94-220. Dartmouth College, 18 July 1994.
- [14] Marshall K. McKusick, William N. Joy, Samuel J. Leffler, and Robert S. Fabry. A fast file system for UNIX. *ACM Transactions on Computer Systems*, 2(3):181–97, August 1984.
- [15] L. Mumment and M. Satyanarayanan. Long Term Distributed File Reference Tracing: Implementation and Experience. CMU-CS-94-213, November 1994.
- [16] Michael N. Nelson, Brent B. Welch, and John K. Ousterhout. Caching in the Sprite network file system. *ACM Transactions on Computer Systems*, 6(1):134–54, February 1988.
- [17] John K. Ousterhout, Hervé Da Costa, David Harrison, John A. Kunze, Mike Kupfer, and James G. Thompson. A trace-driven analysis of the Unix 4.2 BSD file system. *Proceedings of the 10th ACM Symposium on Operating Systems Principles* (Orcas Island WA (USA)). Published as *Operating Systems Review*, 19(5):15–24, December 1985.
- [18] A. L. Narasimha Reddy and James C. Wyllie. I/O Issues in a Multimedia System. *IEEE Computer*, pages 69–74, March 1994.
- [19] Timothy Roscoe. The Structure of a Multi-Service Operating System. Technical report 94-5. Pegasus project, Esprit BRA 6586, March 1994.
- [20] Mendel Rosenblum and John K. Ousterhout. The design and implementation of a log-structured file system. *Proceedings of 13th ACM Symposium on Operating Systems Principles* (Pacific Grove, CA, USA, October 1991). Published as *SIGOPS*, 25(5):1–15. Association for Computing Machinery, October 1991.
- [21] Chris Ruemmler and John Wilkes. An Introduction to Disk Drive Modeling. *IEEE Computer*, pages 17–28, March 1994.
- [22] Chris Ruemmler and John Wilkes. UNIX Disk Access Patterns. *USENIX Technical Conference Proceedings* (San Diego, CA), pages 405–20. USENIX, Winter 1993.
- [23] Russel Sandberg, David Goldberg, Steve Kleiman, Dan Walsh, and Bob Lyon. Design and Implementation of the Sun Network Filesystem. *USENIX Conference Proceedings* (Portland, OR), pages 119–30. USENIX, Summer 1985.
- [24] Small Computer System Interface – 2. American National Standard for Information systems, 23 June 1989. Working draft proposal.
- [25] Margo Seltzer. A comparison of data manager and file system supported transaction processing. Computer Science Div., Department of Electrical Engineering and Computer Science, University of California at Berkeley, January 1990. Obtained from the author.
- [26] Margo Seltzer, Keith Bostic, Marshall Kirk McKusick, and Carl Staelin. An Implementation of a Log-Structured File System for UNIX. *USENIX Technical Conference Proceedings* (San Diego, CA), pages 307–26. USENIX, Winter 1993.
- [27] Margo Seltzer, Keith A. Smith, Hari Balakrishnan, Jacqueline Chang, Sara McMains, and Venkata Padmanabhan. File system logging versus clustering: a performance comparison. *Conference Proceedings of the USENIX 1995 Technical Conference on UNIX and Advanced Computing Systems* (New Orleans), pages 249–64. Usenix Association, 16–20 January 1995.
- [28] Margo Ilene Seltzer. *File system performance and transaction support*. PhD thesis. University of California, 1992.
- [29] Ken Shirriff. Allspice's configuration, 23 March 1995. Private communication.
- [30] Chandramohan A. Thekkath, John Wilkes, and Edward D. Lazowska. Techniques for file system simulation. *Software—Practice and Experience*, 24(11):981–99. John Wiley and Sons Ltd, November 1994.
- [31] John Wilkes, Richard Golding, Carl Staelin, and Tim Sullivan. The HP AutoRAID hierarchical storage system. *Proceedings of 15th Symposium on Operating System Principles*, 1995.

Author information

Peter Bosch graduated with degrees in EE (Bc., 1988), and CS (M.Sc., 1994) from the Univ. of Twente, NL. He has worked since 1991 for the Univ. of Twente as a Ph.D. student. His current work involves research into file-systems and the implementation of a high-performance file-system for the ESPRIT PEGASUS project. His hobbies include traveling and spending evenings in restaurants.

Prof. Sape J. Mullender is chairman of the systems programming and architecture department in the Faculty of Computer Science of the University of Twente in the Netherlands, where he leads the Huygens research project on fault-tolerance, real time, multimedia and security in distributed systems.

Sape Mullender is interested in high-performance distributed computing and the design of scalable fault-tolerant services. He is also concerned about organization and protection in distributed systems that can span a continent. He is a principal designer of the Amoeba distributed operating system.

Predicting File System Actions from Prior Events

Thomas M. Kroeger[†]and Darrell D. E. Long^{‡§}
Department of Computer & Information Sciences
University of California, Santa Cruz

Abstract

We have adapted a multi-order context modeling technique used in the data compression method *Prediction by Partial Match* (PPM) to track sequences of file access events. From this model, we are able to determine file system accesses that have a high probability of occurring as the next event. By prefetching the data for these events, we have transformed an LRU cache into a predictive cache that in our simulations averages 15% more cache hits than LRU. In fact, on average our four-megabyte predictive cache has a higher cache hit rate than a 90 megabyte LRU cache.

1 Introduction

With the rapid increase of processor speeds, file system latency is a critical issue in computer system performance [14]. Standard *Least Recently Used* (LRU) based caching techniques offer some assistance, but by ignoring any relationships that exist between file system events, they fail to make full use of the available information.

We will show that many of the events in a file system are closely related. For example, when a user executes the program **make**, this will often result in accesses to the files **cc**, **as**, and **ld**. Additionally, if we note an access to the files **make** and **makefile** then another sequence of accesses: **program.c**, **program.h**, **stdio.h**, . . . , is likely to occur. As a result,

the file system behaves predictably. Thus a predictive caching algorithm that tracks file system events and notes predictive sequences can exploit such sequences by preloading data before it is required. This increases the cache hit ratio and reduces file system latency.

As in data compression, where a model drives a coder, our predictive cache can be divided into two portions: the model that tracks the sequences of previous events and the selector that uses this information to determine likely future events and prefetch their data. Our model tracks previous file system events through a finite multi-order context modeling technique adapted from the data compression technique *Prediction by Partial Match* (PPM) [2]. This model uses a trie [9] to store sequences of previous file system events and the number of times they have occurred. Our selector examines the most recently seen sequences and the counts of the events that have followed them to determine likely next events. Using these predictions, we augment an LRU cache by prefetching data that are likely to be accessed. The result is a predictive caching algorithm that in our simulations averaged hit ratios better than an LRU cache that is 20 times its size.

The rest of this article is organized as follows: §2 details the method used to model events and select events to prefetch, §3 presents our simulations and results, §4 describes related work, §5 discusses future work, and §6 concludes the paper.

2 Predictive Caching Method

The problem of tracking file system events and the sequences in which they occur is quite similar to the text compression problem of tracking strings of char-

[†]Internet: tmk@cse.ucsc.edu, Telephone (408) 459-4458.

[‡]Internet: darrell@cse.ucsc.edu, Telephone (408) 459-2616.

[§]Supported in part by the Office of Naval Research under Grant N00014-92-J-1807.

acters to model their frequency distributions. The use of data compression modeling techniques for prefetching in operating systems was first investigated Vitter, Krishnan and Curewitz [19, 4]. It was their work that inspired us to adapt PPM’s modeling techniques, to track file system events instead of characters of an alphabet. Using the information tracked by our model we chose a method based on a likelihood threshold to select the events to prefetch.

2.1 Context Modeling

Just as a word in a sentence occurs in a context, a character in a string can be considered to occur in a context. For example, in the string “**object**” the character “t” is said to occur within the context “**objec**”. However, we can also say that “t” occurs within the context “c”, “ec”, “jec”, and “bjec”. The length of a context is termed its *order*. In the example string, “jec” would be considered a third order context for “t”. In text compression these contexts are used to model which characters are likely to be seen next. For example, given that we have seen the context “**object**” it may be likely that the next character will be a space, or possibly an “i”, but it is unlikely that the next character is an “h”. On the other hand, if we only consider the first order context, “t”, then “h” is not so unlikely. Techniques that track multiple contexts of varying orders are termed *Multi-Order Context Models* [2]. To prevent the model from quickly growing beyond available resources, most implementations of a multi order context model limit the highest order tracked to some finite number (m), hence the term *Finite Multi-Order Context Model*.

At every point in the string, the next character can be modeled by the last seen contexts (a set of order 0 through m). For example, take the input string “**objec**” and limit our model to a third order ($m = 3$). The next character can now be described by four contexts $\{ \emptyset, "c", "ec", "jec" \}$. This set of contexts can be thought of as the current state of whatever we are modeling, be it a character input stream or a sequence of file system events. With each new event, the set of new contexts is generated by appending the newly seen event to the end of the contexts that previously modeled our state. If the above set was our current state at time t , and at time $t + 1$ we see the character “t”, our new state

is described by the set $\{ \emptyset, "t", "ct", "ect" \}$. The nature of a context model, where one set of contexts is built from the previous set, makes it well suited for a trie¹ [9], where the children of each node indicate the events that have followed the sequence represented by that node. A resulting property of this trie is that the frequency count for each current context is equal to the sum of its children’s counts plus one². It is from this property that we derive our probability estimate in §2.3.

2.2 Tracking File System Events

In our model contexts are sequences of file system events. To store all the previously seen contexts we use a trie. Each node in this trie contains a specific file system event. Through its path from the root, each node represents a sequence of file system events, or a context, that has been previously seen. Within each node we also keep a count of the number of times this sequence has occurred.

To easily update our model and use it to determine likely future events, we maintain an array of pointers, 0 through m , that indicate the nodes which represent the current contexts (C_0 through C_m). With each new event A , we examine the children of each of the old C_k , searching for a child that represents the event A . If such a child exists, then this sequence (the new C_{k+1}) has occurred before, and is represented by this node’s child, so we set the new C_{k+1} (the $k + 1^{th}$ element of our array) to point to this child and increment its count. If no such child is found, then this is the first time that this sequence has occurred, so we create a child denoting the event A and set the $k + 1^{th}$ element of our array to point to its node. Once we have updated each context in our set of current contexts, we have a new state that describes our file system. Figure 1 extends an example from Bell [2] to illustrate how this trie would develop when given the sequence of

¹ A trie is commonly used as a efficient data structure to hold a dictionary of words. It is based on a tree in which each node contains a specific character. Each node then represents the sequence of characters that can be found by traversing the tree from the root to that node.

² Note that since nodes of order m must have children (which will be leafs), a model of order m requires a trie of order $m + 1$.

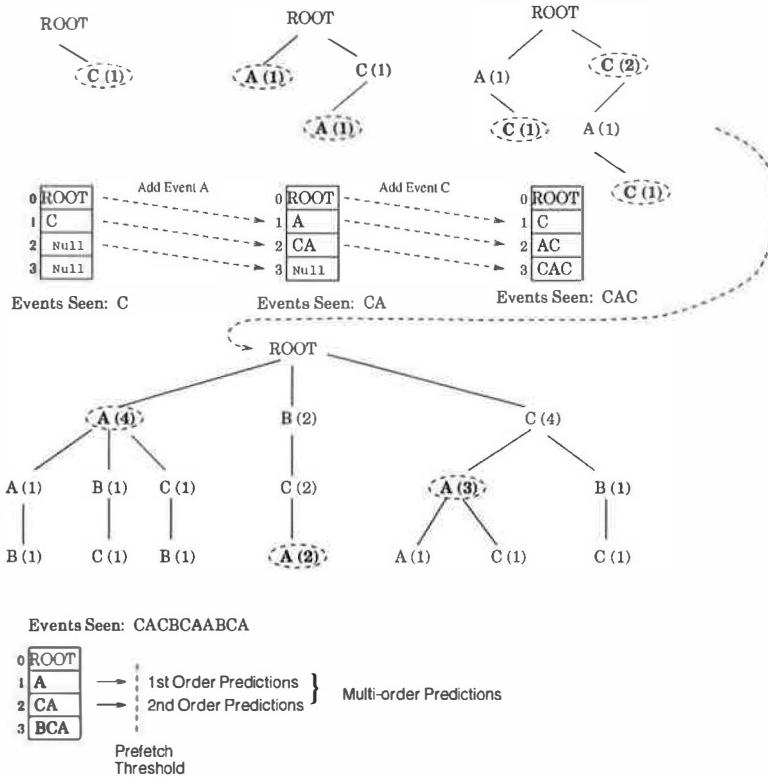


Figure 1: Example tries for the sequence *CACBCAABCA*.

events *CACBCAABCA*. The first three tries show how our model builds from the initial (empty) state. The last trie shows how our model would look after the entire sequence. The current contexts at each stage are indicated by the circled letters.

2.3 Selecting Events to Prefetch

As we generate each of the new contexts, we examine their children to determine how likely they are to be the next event. Using the formula $\text{Count}_{\text{Child}} / (\text{Count}_{\text{Parent}} - 1)$ we generate a maximum-likelihood estimation [17] of the probability of that child's event occurring. We compare this estimate to a probability threshold set as a parameter of our algorithm. If the estimated likelihood is greater than or equal to this threshold, then the data accessed for this event is prefetched into the cache. We evaluate each

context 1 through m independently, resulting in m sets of predictions. The zero order context is a Least Frequently Used (LFU) model and therefore was thought to be of little benefit. Consequently the selector does not examine the zero order context for predictions.

Prefetched data is placed at the front of our cache, and since cache replacement is still LRU, the data will most likely be in the cache for the next several events. The result is that although our cache prefetches based on predictions for what the next event will be, since the prefetched data is likely to be in the cache for more than just the next event, as long as the event occurs before its data is removed from the cache we still avoid a cache miss.

3 Simulation

To simulate the workload of a system, we used file open events from the Sprite file system traces [1]. We chose to consider whole file caching for three reasons. The primary purpose of our work is to avoid the latency of file system accesses; if a whole file can be cached, this reduces the number of transactions with the I/O subsystem on behalf of that file, and in turn reduces the latency of our file system. Whole file caching has been used effectively in several distributed file systems [7, 8, 18]. In a mobile environment the possibility of temporary disconnection and the availability of local storage make whole file caching the best option.

We split the file system traces into eight 24 hour periods lettered A through H. Given the time frame and environment under which these traces were generated, we chose a cache size of four megabytes as a reasonable size for our initial tests. After developing an understanding of how the various parameters effected performance, we explored our model’s performance for cache sizes up to 256 megabytes.³

3.1 Prefetch Threshold

Our first goal was to examine which probability thresholds would result in the best hit ratios. Figure 2 shows how our hit ratio varied as the threshold settings ranged from a probability of 0.001 to 0.25. From this graph we can see that a setting in the region of 0.05 to 0.1 will offer the best performance. From Griffioen and Appleton’s work [6] and our earlier work, we expected this setting to be quite low. Even so, it is surprising that such an aggressive prefetch threshold produced the best results.

To explain this, we first consider that each trace is comprised of over 10,000 distinct files. Since each of these files can be a child to any node, the tree we build will become very wide. Since the count for each parent is the sum of the counts for its children, such a wide tree would result in parents with much higher counts than their individual children. It follows that the parent count divided by the count of children would be

³For readability our graphs only show cache sizes up to 128 megabytes.

rather low even for children that frequently follow their parent.

For settings greater than 0.025, performance does not change radically with minor variations. However for settings below 0.025 we see a sharp drop in performance as a result of prefetching too many files. Thus we can say that this algorithm is stable for settings of the probability threshold that are greater than 0.025.

3.2 Number of Files Prefetched

We were initially concerned that these low threshold settings might have resulted in prefetching an impractical number of files, but this is not the case. In fact, for a probability threshold of 0.075 the average number of files prefetched per open event ranged from 0.21 to 1.10 files. Figure 3 shows how the average number of prefetches varied for the same settings of probability threshold used in §3.1. This graph shows that for extremely low threshold settings, less than 0.025, the number of files prefetched quickly becomes prohibitive. However, for settings in the region of 0.05–0.1, the average number of files prefetched would not impose an excessive load.

3.3 Maximum Order of the Model

To see how much benefit was gained from each order of modeling, we simulated models of order ranging from zero through four. Since we ignore the predictions of the zero order model, a zero order cache does not prefetch, and is therefore equivalent to an LRU cache. Figure 4 shows how performance varied over changes in model order. While we expected to gain mostly from the first and second orders, the second order improved performance more than we had expected, while fourth and higher orders appeared to offer negligible improvements. We hypothesize that the significant increase from the second order model comes from its ability to detect the combination of some frequently used file (*e.g.* `make` or `xinit`) and a task-specific file (*e.g.* `Makefile` or `.xinitrc`).

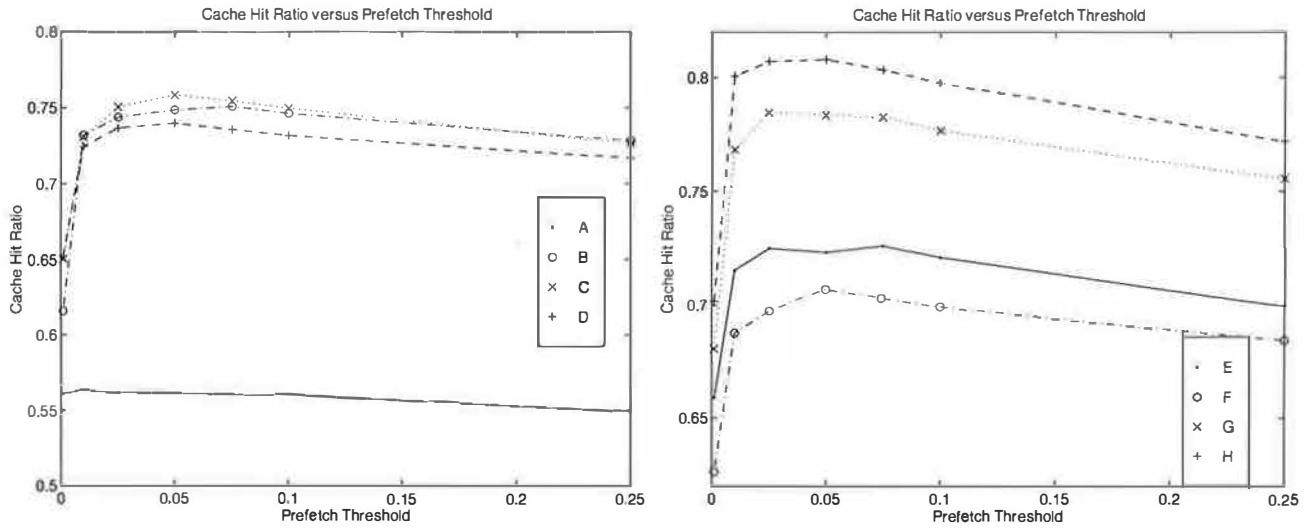


Figure 2: Cache hits versus prefetch threshold (cache size 4 megabytes, second order, threshold settings 0.001, 0.01, 0.025, 0.05, 0.075, 0.1 and 0.25).

3.4 Improvements Over LRU

With a firm understanding for the appropriate parameters of our model, we compared our predictive cache to an LRU cache. For this comparison, both caches simulated four megabytes of cache memory. Our predictive cache extended to the second order and prefetched at a conservative threshold of 0.1. Table 1 shows the results of our simulations. Our predictive cache clearly offered significant improvements over the performance of LRU on all eight traces, averaging 15% more cache hits than LRU and, in the case of trace E as much as 22% more.

3.5 Cache Size

One key concern we had was whether the benefit from our predictive cache would quickly diminish as the size of our cache grew. In order to investigate this we simulated an LRU cache and our predictive cache for varying cache sizes up to 256 megabytes. Figures 5 and 6 show how the cache hit ratios varied as we increased the cache size. From these graphs it is clear that tracking file system actions offers a performance gain that will not easily be overcome by increasing

the size of an LRU cache. For example, on average it would require a 90 megabyte LRU cache to match the performance of a 4 megabyte predictive cache.

3.6 Model Memory Requirements

The amount of memory required in our current implementation is directly proportional to the number of nodes in our trie. On average a second order model required 238,200 nodes. Since our implementation required 16 bytes per node, the memory required by a third order model should be well under four megabytes. While this model takes almost as much memory as the cache it models, we note that this additional four megabytes seems negligible when compared to the additional 86 megabytes that would be required for an LRU cache to see equivalent performance. It should also be noted that model size is independent of cache size, therefore a 32 megabyte predictive cache would still require less than four megabytes of model space.

Additionally, in our initial implementation we have made no effort to efficiently use memory in our model. We intend to expand on methods used successfully in the compression technique DAFC [13], to limit the

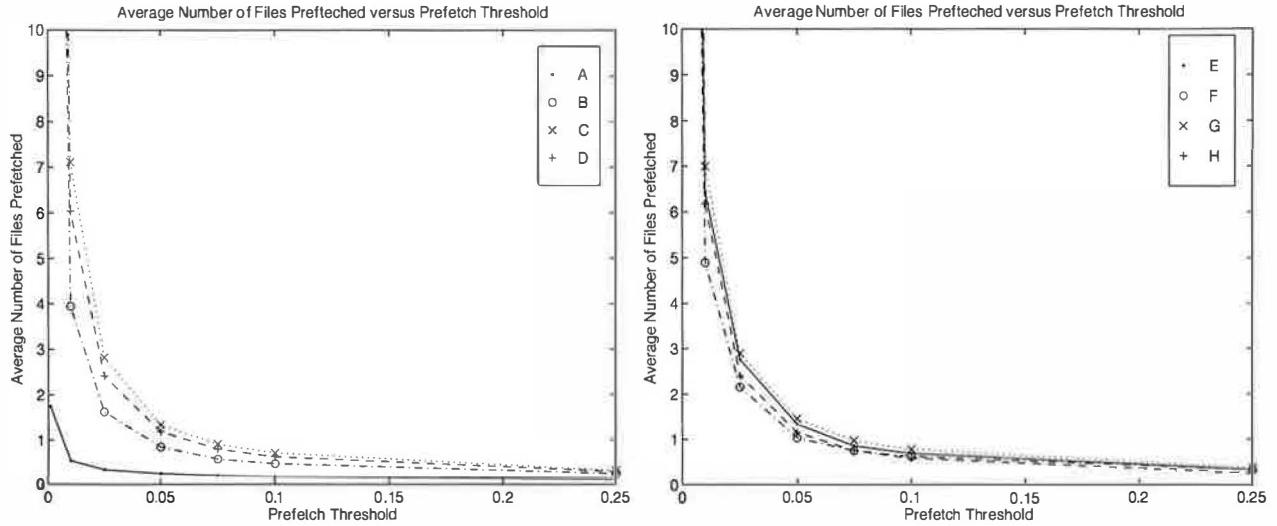


Figure 3: Average number of files prefetched per open event versus prefetch threshold (cache size 4 megabyte, second order, threshold settings 0.001, 0.01, 0.025, 0.05, 0.075, 0.1 and 0.25).

Trace	A	B	C	D	E	F	G	H
Predictive	56.1%	74.6%	75.0%	73.2%	77.1%	70.0%	77.7%	79.8%
LRU	48.5%	59.7%	59.8%	57.2%	54.9%	54.0%	58.4%	68.8%
Improvement	7.6%	14.9%	15.2%	16.0%	22.2%	16.0%	19.3%	11.0%

Table 1: Hit ratios for LRU and predictive caches (cache size 4 megabytes, second order model, threshold 0.1).

number of children that each node in our context model has, and to periodically refresh parts of the set of children by releasing links taken up by less frequently seen children. We expect that this modification will not only significantly reduce the memory requirements of our model, but will also allow it to adapt to patterns of local activity. Limiting the number of children a node can have will also ensure that the time required to update our model and predict new accesses is limited to a constant factor.

4 Related Work

Vitter, Krishnan and Curewitz were the first to examine the use of compression modeling techniques to track events and prefetch items [19]. They prove that such techniques converge to an optimal online algorithm.

They go on to test this work for memory access patterns [4] in an object oriented database and a CAD system. They deal with the large model size by paging portions of the model to secondary memory, and show that this can be done with negligible effect on performance. Additionally they suggest that such methods could have great success within a variety of other applications such as hyper-text. Our work adapts PPM in a different manner. We avoid the use of vine pointers [2, 10] and instead keep an array of the current contexts. Their method of selection for prefetching (choosing the n most probable items, where n is a parameter of their method) differs from the threshold based method we use. Lastly, the problem domain we examine (file systems access patterns) differs from that which they have worked under (virtual memory access patterns).

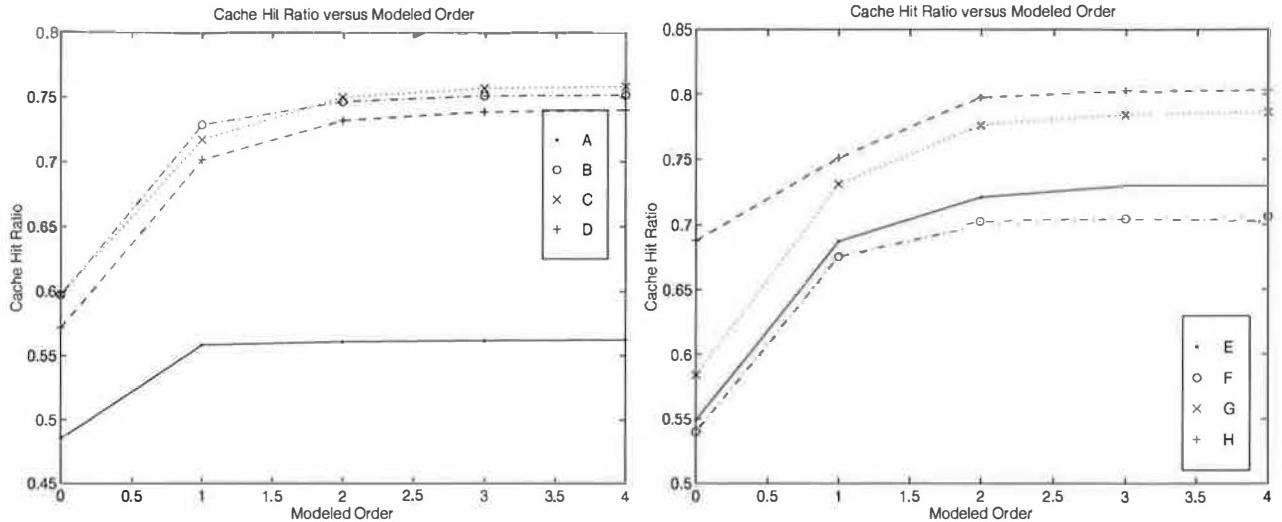


Figure 4: Cache hit ratio versus model order (cache size 4 megabytes, prefetch threshold 0.1).

Within the domain of file systems, Griffioen and Appleton [6] have worked to developed a predictive model that for each file accumulates frequency counts of all files that are accessed in a window after the first. These frequency counts are then used to drive a prefetching cache. Their prediction model differs from ours in that they look at more than just the next event. Additionally, they only consider a first order model. Nevertheless, the method of prefetch selection based on a probability threshold was first presented in their work.

Kuennen, Popek, and Reiher [12] have done extensive work analyzing the behavior of file system requests for various mobile environments with the intent of developing a prefetching system that would predict needed files and cache them locally. Their work has concluded that such a predictive caching system has promise to be effective for a wide variety of environments. Kuennen has extended this work [11], developing the concept of a *Semantic Distance*, and using this to determine groupings of files that should be kept on local disks for mobile computers.

Patterson, *et al.* [16, 15], have modified a compiler and file system to implement a method called *Transparent Informed Prefetching* where applications inform

the file system which files to prefetch. While this method can offer significant improvements in throughput, it is dependent on the applications ability to know its future actions. For example `cc` would only know which header files it would need once it had read in the line `#include <stdio.h>`, while our predictive model could notice that every access to `program.c` causes an access to `stdio.h`. Finally, such an application specific method would not be able to make use of any relationships that exist across applications (*e.g.* between `make` and `cc`).

Cao *et al.* [3] have approached this problem from a unique perspective, by examining what characteristics an off-line prefetching technique would require to be successful.

5 Future Work

The following items are intended as areas of future exploration:

Trie memory requirements – Our current implementation was designed as a proof of concept without concern for the memory usage of our predictive trie. Both Griffioen [5] and Curewitz [4] successfully ad-

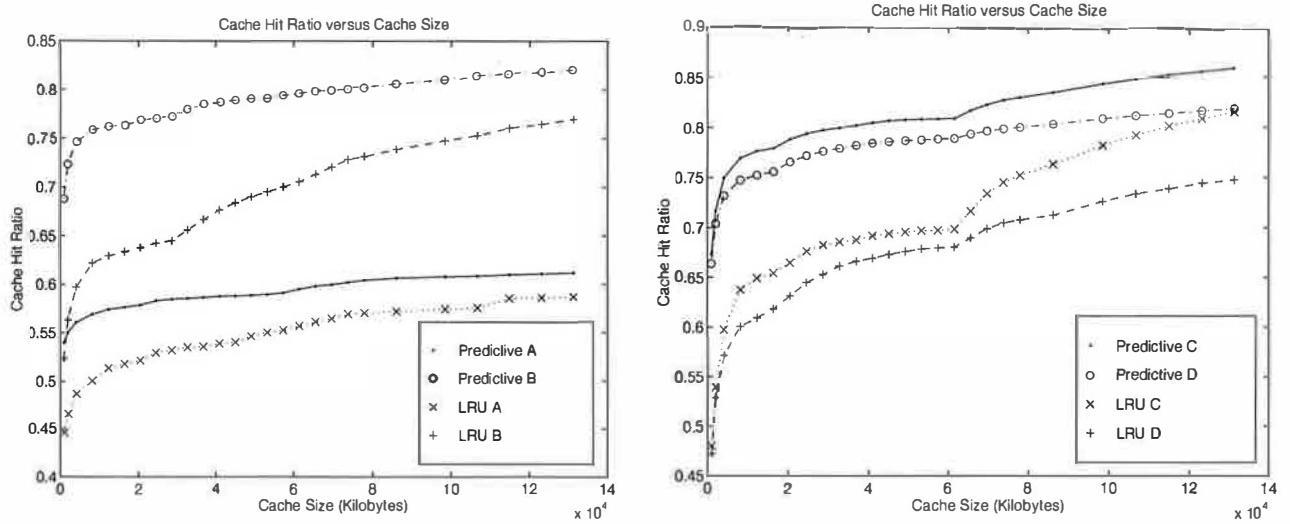


Figure 5: Cache hit ratio versus cache size for both predictive cache and LRU (cache sizes 1–128 megabytes).

dress this issue. We hope to expand on their work, in conjunction with data compression techniques. We expect that our predictive cache will see significantly improved efficiency after we implement a fixed limit on the number of children each node can have. Additionally, methods that give more weight to more recently seen patterns and purge old informations not only reduce the model size, but also can improve performance.

Predicting from grandparents – It is quite possible for one file open event to be the cause of two future events that occur closely. As a result, the order of these two events may vary. For example, an open of file *B* causes opens to files *A* and *C*, so we would have one of two resulting sequences *BAC* or *BCA*. We intend to investigate using all the descendants of a context to predict possible variations in the ordering of events. Using the final tree from Figure 1, if we see an open of file *B*, then we would not only prefetch file *C* but also file *A* as well. Such a forward-looking prediction would enable us to avoid cache misses for the sequence *BAC* as well as *BCA*.

Modifications to the prefetching algorithm – Curewitz's [4] approach to prefetching is to select the n most likely items (where n is a parameter of the model). We intend to investigate a combination of this

method and the threshold based selection that we have used by placing a limit on the number of files that can be prefetched. We also intend to investigate the effect of having different threshold settings for each order of the model.

Predictive replacement – While our cache is based on a predictive model to prefetch files, it still uses LRU to determine which files to expel from the cache when space is needed. Using our predictive model to determine cache replacements may offer further improvements in performance.

Read wait times – While cache hit ratios are commonly used to measure the performance of a caching algorithm, we are mostly concerned with the amount of time that is spent waiting for file access events to complete. Our intent is to extend our simulation to include read-wait times allowing us to account for the additional I/O load generated by prefetching.

Selection by cost comparison – Finally we intend to explore a prefetching selection model that uses the estimated probabilities of our model, in conjunction with other factors, such as memory pressure and file size, to estimate the cost, in terms of read-wait times. These estimates would be used in each case to decide if it was more beneficial to prefetch or not. Our hope is

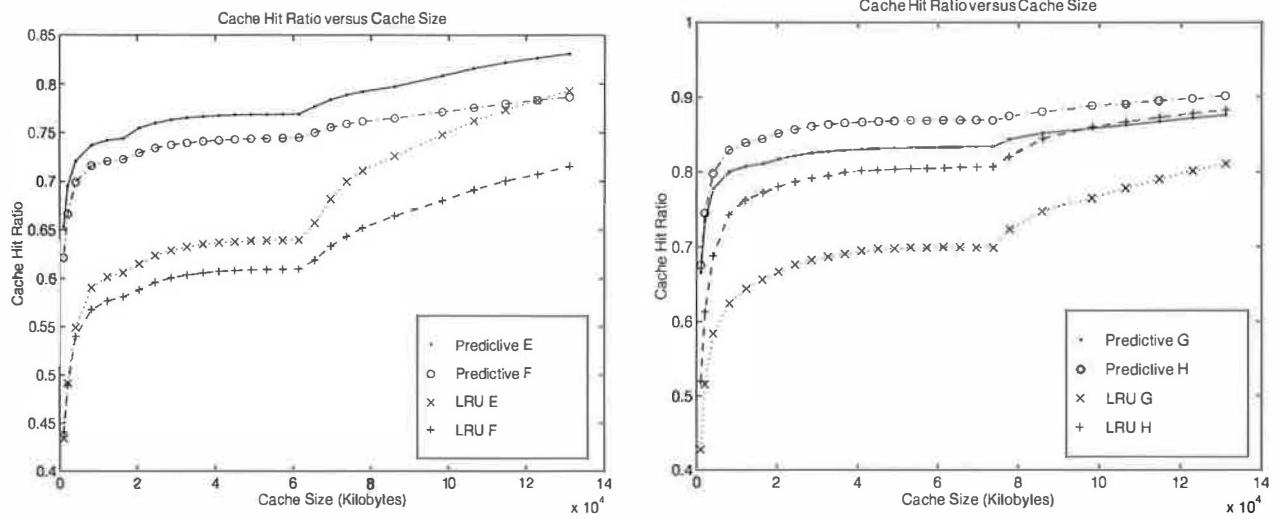


Figure 6: Cache hit ratio versus cache size for both predictive cache and LRU (cache sizes 1–128 megabytes).

that such a parameterless selection method will adjust to changing file system behavior.

6 Conclusions

Our prediction model shows how file system events can successfully be modeled with the multi-order techniques used in PPM. Through trace driven simulations, we have demonstrated that this model can be used effectively to predict future file system events. From our ability to effectively predict future events based on previous file system events, we have shown strong empirical evidence that there exists consistent and exploitable relationships between file accesses.

As the I/O gap widens due to advances in processor design, file system latency will become a greater hindrance to overall performance. By exploiting the highly related nature of file system events we can use methods such as predictive caching to reduce file system latency and improve overall system performance.

7 Acknowledgments

This line of investigation was originally inspired by the comments of Dr. J. Ousterhout regarding relationships between files. We are grateful to Prof. M. Baker for her support in working with the Sprite file system traces, B. Sherrod for his insightful comments and continued support, R. Appleton for sharing advance copies of his work and his experiences, T. Van Vleck for his many comments and his idea of a parameterless selection technique, Dr. L. F. Cabrera for his input and guidance, and to the many other people who offered their time and comments.

References

- [1] M. G. Baker, J. H. Hartman, M. D. Kupfer, K. W. Shirriff, and J. K. Ousterhout. Measurements of a Distributed File System. *Proceedings of 13th Symposium on Operating Systems Principles*, pages 198–212. ACM, Oct. 1991.
- [2] T. C. Bell, J. G. Cleary, and I. H. Witten. *Text Compression*. Prentice Hall, 1990.

- [3] Pei Cao, Edward W. Felten, Anna R. Karlin, and Kai Li. A Study of Integrated Prefetching and Caching Strategies. *Proceedings of the 1995 SIGMETRICS*. ACM, 1995.
- [4] K. M. Curewitz, P. Krishnan, and J. S. Vitter. Practical Prefetching via Data Compression. *SIGMOD Record*, 22(2):257–266. ACM, Jun. 1993.
- [5] J. Griffioen and R. Appleton. Reducing File System Latency Using a Predictive Approach. *Proceedings of USENIX Summer Technical Conference*, pages 197–207. USENIX, Jun. 1994.
- [6] J. Griffioen and R. Appleton. Performance Measurements of Automatic Prefetching. *Parallel and Distributed Computing Systems*, pages 165–170. IEEE, Sept. 1995.
- [7] J. H. Howard, M. L. Kazar, S. G. Menees, D. A. Nichols, M. Satyanarayanan, R. N. Sidebotham, and M. J. West. Scale and Performance in a Distributed File System. *Transactions on Computer Systems*, 6(1):51–81. ACM, Feb. 1988.
- [8] J. J. Kistler and M. Satyanarayanan. Disconnected Operation in the Coda File System. *Proceedings of the 13th Symposium on Operating Systems Principles*, pages 213–25. ACM, Oct. 1991.
- [9] D. E. Knuth. *Sorting and Searching*, volume 3 of *The Art of Computer Programming*. Addison-Wesley, Reading, MA, 1973.
- [10] P. Krishnan. *Online Prediction Algorithms for Databases and Operating Systems*. PhD thesis. Brown University, May 1995.
- [11] G. Kuenneng. The Design of the Seer Predictive Caching System. *Workshop on Mobile Computing Systems and Applications*, pages 37–43. IEEE, Dec. 1994.
- [12] G. Kuenneng, G. J. Popek, and P. Reiher. An Analysis of Trace Data for Predictive File Caching in Mobile Computing. *Proceedings of USENIX Summer Technical Conference*, pages 291–303. USENIX, 1994.
- [13] G. G. Langdon and J. J. Rissanen. A Doubly-Adaptive File Compression Algorithm. *IEEE Transactions on Communications*, COM-31(11):1253–1255, Nov. 83.
- [14] J. Ousterhout. Why Aren't Operating Systems Getting Faster as Fast as Hardware? *Proceedings of USENIX Summer Technical Conference*, pages 247–56. USENIX, 1990.
- [15] H. Patterson, G. Gibson, E. Ginting, D. Stodolsky, and J. Zelenka. Transparent Informed Prefecting. *Proceedings of 13th Symposium on Operating Systems Principles*. ACM, Dec. 1995.
- [16] H. Patterson, G. Gibson, and M. Satyanarayanan. A Status Report on Research in Transparent Informed Prefecting. *Operating Systems Review*, 27(2):21–34. ACM, Apr. 1993.
- [17] K. Trivedi. *Probability & Statistics with Reliability, Queueing, and Computer Science Applications*. Prentice Hall, 1982.
- [18] R. van Renesse, A. S. Tanenbaum, and Annita Wilschuts. The Design of a High-Performance File Server. *Proceedings of the 9th International Conference on Distributed Computing System*, pages 22–27. IEEE Computer Society Press., 1989.
- [19] J. S. Vitter and P. Krishnan. Optimal Prefetching via Data Compression. *Proceedings 32nd Annual Symposium on Foundations of Computer Science*, pages 121–130. IEEE Comput. Soc. Press, Oct. 1991.

Transparent Fault Tolerance for Parallel Applications on Networks of Workstations

*Daniel J. Scales** and *Monica S. Lam*
Computer Systems Laboratory
Stanford University
Stanford, CA 94305

Abstract

This paper describes a new method for providing transparent fault tolerance for parallel applications on a network of workstations. We have designed our method in the context of shared object system called SAM, a portable run-time system which provides a global name space and automatic caching of shared data. SAM incorporates a novel design intended to address the problem of the high communication overheads in distributed memory environments and is implemented on a variety of distributed memory platforms. Our fundamental approach to providing fault tolerance is to ensure the replication of all data on more than one workstation using the dynamic caching already provided by SAM. The replicated data is accessible to the local processor like other cached data, making access to shared data faster and potentially offsetting some of the fault tolerance overhead. In addition, our method uses information available in SAM applications on how processes access shared data to enable several optimizations which reduce the fault-tolerance overhead.

We have built an implementation of our fault-tolerance method in SAM for heterogeneous networks of workstations running PVM3. In this paper, we present our fault-tolerance method and describe its implementation in detail. We give performance results and overhead numbers for several large SAM applications running on a cluster of Alpha workstations connected by an ATM network. Our method is successful in providing transparent fault tolerance for parallel applications running on a network of workstations and is unique in requiring no global synchronizations and no disk operations to a reliable file server.

*Author's current address: Digital Equipment Corporation Western Research Laboratory, 250 University Ave., Palo Alto, CA 94301.
This research was supported in part by DARPA contract DABT63-91-K-0003.

1 Introduction

Networks of workstations linked by high-speed interconnects such as ATM networks look quite attractive as platforms for running parallel applications, especially since many institutions already have numerous workstations on employees' desks or in common areas that are often idle. To date, networks of workstations have been used for only a limited range of parallel applications, because of issues of programmability and efficiency. The basic message-passing primitives provided by systems such as PVM [18] support only a very low-level programming interface and are difficult to use for programming applications with complex communication patterns. Systems that support a global shared-memory model in software greatly ease programming of parallel applications, but can encounter efficiency problems because of the high cost of communication in workstation environments.

In addition, programmers must deal with the possibility that individual processes or workstations participating in a parallel computation may fail. Workstations are often administered by the individual owners or reside in public areas. It is therefore not uncommon for workstations to be rebooted without notice. Because workstations are a shared resource, processes on a workstation may fail because they are explicitly killed by another user or exceed the resource limits of the workstation. An applications programmer using a cluster of workstations must either accept the loss in performance that results when an application run must be restarted after a failure, or attempt to deal explicitly with the possibility of failure in the application code.

SAM is a run-time system for distributed memory machines and networks of workstations that eases programming by providing a global name space and dynamic caching of shared data in software at the level of user-defined types (or *objects*) [15, 16]. SAM incorporates a novel design that retains many of the ef-

ficient properties of message passing and minimizes communication. The basic approach in SAM is to require the programmer to designate the way in which data will be accessed, thus allowing communication for synchronization and data access to be combined. Producer/consumer relationships are expressed by accesses to single-assignment values, and mutual exclusion constraints are represented by access to data items called accumulators. SAM currently runs on the CM-5, Intel Paragon, IBM SP2, and heterogeneous networks of workstations. Experience with SAM has shown that it significantly eases the programming of complex applications and allows the programmer to achieve good performance for these applications on a variety of distributed memory platforms.

This paper describes a method of providing fault tolerance in software for long-running parallel applications on networks of workstations written using SAM. Our method recovers from the common case of a small number of processors that fail by halting; it does not handle a global system failure or failures in which processors operate incorrectly. It is desirable that a fault-tolerance mechanism have transparency, low overhead, portability, and scalability. By transparency, we mean that the fault-tolerance mechanism can automatically be used for applications without any additional programming by the user. The fault-tolerance mechanism should have small overhead, so that the performance benefits of running an application across many workstations is not significantly reduced by the cost of providing fault tolerance. By portability, we mean that the fault-tolerance mechanism can be used on a variety of kinds of workstations without changes to the basic system software. Finally, the mechanism should be inherently scalable and must therefore minimize the need for global communication and synchronization.

Dynamic caching of shared data is an important part of the SAM design which makes it possible to exploit the data locality in parallel applications. Our fundamental method of providing fault tolerance is to use the dynamic caching functionality already provided by SAM to ensure, at checkpoints, that all shared data is replicated on more than one *host* (workstation). Our method thus provides fault tolerance without doing expensive writes to disk and does not require a common file server. Our integration of the fault tolerance method with the SAM system provides advantages of simplicity and efficiency. The implementation of the fault tolerance is simplified by using the existing caching mechanism. Because the replicated data is accessible to the local host like all other cached data, the apparent overhead of the fault tolerance method may be reduced by faster application access to shared data. In addition, though our method is applicable to any shared object

system with caching, it uses information available about data accesses in SAM applications to reduce the fault tolerance overhead.

Our method runs on an arbitrary collection of workstations with standard operating systems and transparently recovers from the common case of a single (or a small number of) host failures without any user intervention. Only the process on the failed host must be restarted and redo some computation; other processes continue executing without any rollback. It also avoids global synchronizations among all the workstations. The main disadvantage of our method is that the amount of fault-tolerance overhead depends on the communication pattern of the application and can potentially be large. In addition, because of its technique of replicating data in the memory of other hosts, our method may increase the memory requirements of an application on each host.

We have implemented our method in the SAM system for heterogeneous collections of workstations, using PVM3 as the underlying message-passing layer. PVM3 provides portability to a wide variety of processor architectures and interconnects, and supports the basic functionality for detecting when process and host failures occur. Our implementation automatically provides transparent fault tolerance for all SAM applications in any such environment. In this paper, we describe our method in detail and provide performance results for several large parallel applications running on a network of workstations and using our fault tolerance method.

In the next section, we give a brief overview of the SAM system. We then describe the basic software approaches that have been taken for making parallel applications fault-tolerant. We describe the key ideas of our approach and its implementation in detail. We then provide performance results for several SAM applications running in parallel on workstations connected by an ATM network and analyze the overhead of our method for fault tolerance. Finally, we compare with other software distributed shared memory systems that have provided fault tolerance and conclude.

2 SAM Overview

In this section, we present a brief overview of SAM mainly via several examples. SAM is implemented as preprocessor and run-time system for C programs, but could be modified to work with FORTRAN-90 and C++ applications as well. A more detailed description is provided in [15, 16].

SAM is a software distributed shared memory system that provides a global name space for accessing shared

Shared Address Space	Distributed Address Space + SAM
1) Mutual Exclusion	
<pre>wait (lock) a[1] = ...; a[34] = ...; signal (lock)</pre>	<pre>begin_update_accum(a) a[1] = ...; a[34] = ...; end_update_accum(a)</pre>
2) Producer/consumer	
<pre>i=i+1 j=j+1 allocate a_i wait(flag_j) a_i = = a_j signal(flag_i)</pre>	<pre>i=i+1 j=j+1 begin_create_value(a_i) a_i = ... end_create_value(a_i)</pre>
3) Finite buffer (size 4)	
<pre>i=i+1 mod 4 j=j+1 mod 4 wait(buf_i) wait(flag_j) a_i = = a_j signal(flag_i)</pre>	<pre>i=i+1 j=j+1 begin_rename_value(a_{i-4}→a_i) a_i = ... end_rename_value(a_i)</pre>

Figure 1: Examples of Creating and Accessing Data

objects on distributed memory machines. In SAM, all shared data is communicated at the level of user-defined types (objects) and is represented by either a *value* or an *accumulator*. (SAM deals only with the management and communication of shared data; data that are completely local to a process can be managed by any appropriate method.) In Figure 1, we show several common idioms, as they would be expressed using semaphores on a shared address space machine and using SAM primitives. While the SAM primitives differ from the typical primitives on a shared memory machine, the complexity of expressing the examples using the two models is quite similar.

In the first example, mutual exclusion is required to protect updates to shared data. In SAM, an accumulator is used to represent a piece of data that is to be updated a number of times, and whose final value is independent of the order in which the updates occurs. SAM automatically migrates the accumulator between processes as necessary and ensures that a process does not access the accumulator until mutual exclusion is obtained. Updates to an accumulator must be encapsulated by the SAM primitives `begin_update_accum` and `end_update_accum`. The call to `begin_update_accum` returns a pointer by which the accumulator can be accessed. SAM supports the idiom of chaotic computation via primitives which provide read access to a “recent” value of the accumulator, which is not guaranteed to be the most current value of the accumulator. SAM maintains a

cache in each process of versions of accumulators that have been recently accessed and may be able to satisfy the chaotic request locally without communication.

In the second example, a consumer (right column) accesses data created by a producer (left column). In SAM, a value provides producer/consumer synchronization. Values have a single-assignment semantics: a value is atomically created once its initial contents are set and is henceforth immutable. The code to create a value, which may include arbitrary updates to different components of the value, is encapsulated by a pair of primitives `begin_create_value` and `end_create_value`. Similarly, code accessing a value is encapsulated by the primitives `begin_use_value` and `end_use_value`. A process that attempts to access a value will automatically be suspended until the value is created and has been brought to the local process. Conversely, an access will succeed immediately if the value is already available in the local process, returning a pointer to the local copy.

SAM maintains copies of values fetched from remote processes in local main memory in the form of a cache. Because all values have distinct names and are immutable, there is no consistency problem associated with maintaining this cache. SAM automatically frees up local copies that are not in use in an LRU fashion when the cache memory becomes filled. SAM must ensure that at least one copy of a value is maintained in the system, until it can determine that there will not be any other processes that will need to access the value.

The SAM programmer provides this information by specifying the number of accesses to the value that will occur or explicitly indicating when all accesses to the value have occurred.

In the third example, a consumer accesses a sequence of values created by a producer through a limited-sized buffer. To avoid memory usage problems that are associated with single-assignment values, SAM allows one value to reuse the storage of another value via the `begin_rename_value` primitive. This primitive provides the necessary synchronization to ensure that the producer does not reuse the storage of a value before the consumer has accessed it. In this way, imperative data objects are easily represented in SAM via a sequence of values which can all share the same storage. SAM also allows a value to be converted to an accumulator and vice versa.

The creator of a value or an accumulator must specify the data type of the new object. With the help of a preprocessor, SAM uses this type information to allocate space for, pack (for sending in a message), unpack, and free the storage of the data. The preprocessor can handle complex C data types, including structures, arrays, and types that contain pointers and therefore are not necessarily stored in one contiguous block in memory. In heterogeneous environments, SAM also handles any necessary data conversion between dissimilar machines. Data is always transmitted in units of whole objects.

SAM provides two additional mechanisms for optimizing communication. First, it provides an operation for producers to *push* data to consumers. Second, it provides operations to do asynchronous fetches of objects, thus allowing optimizations such as prefetching in which communication is overlapped with computation or other communication. Each of these optimizations is well integrated into SAM's shared memory model.

3 Software Approaches for Tolerating Failures

In this section, we describe some of the common software methods for tolerating failures. We consider parallel applications in which a number of processes cooperate in doing a computation. To achieve parallelism, each process typically runs on a different host (workstation), so we will assume that there is one process on each host.¹ Each process may be described as going through a series of states during its lifetime, which is

¹In a number of our applications, each process is actually executing several tasks sharing the same address space. Our fault-tolerance mechanism is unaffected by the use of these light-weight tasks, except for the existence of multiple stacks.

determined by its initial input parameters, the program code it is executing, and communications with other processes. Most software fault tolerance mechanisms operate by maintaining information about the state of each process in a parallel execution. When a process fails, the mechanism restarts the failed process (and potentially some of the other processes) in a previous state so that the parallel application can continue executing. The fault tolerance mechanism must ensure that the states of the restarted processes are consistent with each other and any remaining processes, so that the processes execute the remainder of the parallel application correctly (i.e. in the same way as some failure-free execution of the application.)

The most common mechanisms for allowing restoration of the state of a process are *checkpointing* and *logging*. A checkpointing mechanism saves essentially the entire state of the process at one point in time. A logging mechanism saves information about changes to the state of the process and may be viewed as an incremental form of checkpointing. The checkpointing or logging information is most commonly saved to disk.

Checkpointing methods for providing fault tolerance in software for parallel applications may be divided broadly into two classes: *consistent checkpointing* methods and *independent checkpointing* methods. We illustrate these methods in Figure 2, where the arrows indicate communication between processes P1, P2, and P3, and the heavy horizontal bars indicate checkpoints. Figure 2(a) shows *consistent checkpointing* [6, 11], in which all processes periodically synchronize and checkpoint their state simultaneously. This method requires a global synchronization that ensures that all processes are in a consistent state (in which all messages sent by one process have been received by the destination process). Then each process saves its entire state (processor registers and its entire virtual address space), typically to disk. If a failure occurs, then the computation is resumed by restoring *every* process to its state at the last checkpoint (as indicated by the dotted line) and restarting execution. The advantage of this approach is that it can recover from any number of host failures, if the checkpoint information is saved to reliable, nonvolatile storage. Also, when failures are very infrequent, the fault-tolerance overhead can be made very low by making the interval between checkpoints very large. The disadvantages are that global communication between all the hosts is required during a checkpoint, and the states of all processes are rolled back to the last checkpoint during a recovery.

Independent checkpointing methods avoid global synchronizations and allow the state of each process to be checkpointed or logged independently. To ensure that a process can be restarted from its last check-

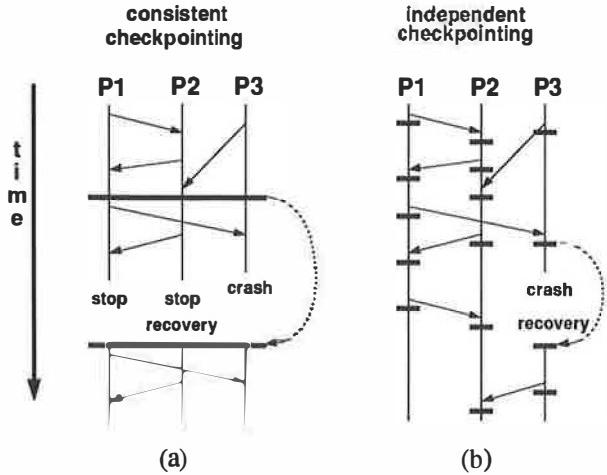


Figure 2: Consistent (a) and Independent (b) Checkpointing

point without affecting other processes, independent checkpointing methods generally require that each process saves its state whenever it communicates with other processes. The independent checkpointing methods divide into two classes, optimistic and pessimistic methods. In the pessimistic method [4], when a process communicates with another process, it atomically checkpoints or logs its incremental state change with the communication. It is then guaranteed that the process can be restarted from its last checkpoint or last set of logging records without requiring any retransmission of communications from other processes and without sending out a duplicate of any communication. The process can therefore be recovered without affecting any other processes. Figure 2(b) illustrates the use of pessimistic independent checkpointing to recover from the crash of a host. The advantage of this approach is that no global synchronization is required, and only the work of the failed process is rolled back. The disadvantage is that fault-tolerance overhead depends on the pattern of communication and can be large.

In the optimistic method [9, 17], a process also checkpoints or logs information when it communicates with another process, but the checkpointing or logging is not required to occur atomically with the communication. Because of this fact, it is not guaranteed that a process can be recovered without affecting the state of other processes. Instead, it may be necessary to rollback the state of other processes which have processed information sent by the process that failed. Dependence information is maintained by all processes to determine which processes need to be rolled back during a recovery.

4 Our Approach

Our approach applies the concepts of pessimistic, independent checkpointing to the SAM system. That is, when a process fails, the recovery procedure involves only restarting that process at a previous state (on the same host or a different host). All other processes proceed normally without rollback. The failed process appears to the other processes during the failure and recovery as if it has become very slow, but otherwise does not affect their execution. However, unlike a naive independent checkpointing approach, our method does not require a checkpoint every time a process is involved in a communication; based on information available about data accesses in SAM applications, it requires checkpoints for only a subset of the communications. In addition, we take the novel approach of checkpointing the state of a process by ensuring that its data is replicated to other processes using the caching mechanism of SAM, instead of saving its data to disk. Using information maintained by SAM, we reduce the amount of state that must be saved by only replicating the objects that have changed since the last checkpoint. In the following sections, we describe our approach in detail. We first describe when a process must do a checkpoint and what state information must be preserved at each checkpoint. We next describe the memory management of data that has been replicated for fault-tolerance purposes. Then we give step-by-step descriptions of the checkpointing procedure and the procedure for recovering from a failure.

4.1 Determining When To Checkpoint

During a parallel execution, each application process communicates with and affects the execution of the other processes. A process restarted after a failure must interact with the other processes in the same way as if the process never failed; otherwise, inappropriate messages sent by the restarted process will cause other processes to execute incorrectly. In a system such as SAM with a shared-memory model, one process can affect the execution of another process only by creating or modifying shared data which is accessed by the other process. A restarted process will therefore interact with other processes as if it never failed if it satisfies the following condition: any shared data produced by the original process that was communicated to other processes (and is still accessible) has the same contents in the restarted process. This condition is sufficient for ensuring a proper execution and is in general necessary, since processes may potentially access any data created or modified by the original process. Our fault-tolerance method must checkpoint sufficiently so that,

at any point in the execution of a process, it can restart the process from a checkpointed state that satisfies the above condition.

The concept of a reexecutable operation is important in deciding when checkpoints are necessary. Suppose an operation is executed when a process is in a particular state and then the process fails. The operation is *reexecutable* if the operation is guaranteed to produce identical effects (on the current process and other processes) when executed from the same state in a restarted process. Any operation which only involves the local process is inherently reexecutable. However, operations that interact with sources external to the process, such as other processes or the operating system, may not be reexecutable. In parallel applications, most operations that are not reexecutable result from communication with other processes. For example, message sends in a message-passing program are not reexecutable, since the restarted process will send out a duplicate message. Similarly, message receives are not reexecutable, since the restarted process will wait for a message that the original process already received.

In the SAM system, a process only interacts with other processes by creating, modifying, or accessing shared data. The creation of a single-assignment value is reexecutable, because a value is not changed once created, and the restarted process will recreate the value with the same contents as in the original process. Similarly, accessing a value is a reexecutable operation, because any value that was accessed by the original process is guaranteed to have the same contents when the restarted process accesses the value again. Conversely, creating or updating an accumulator is not a reexecutable operation, because the accumulator may be modified by another process between the time it created or updated by the original process and the time when the restarted process repeats the operation.

If a process uses the results of an operation that is not reexecutable in creating or modifying a shared data object and there is no checkpoint between that operation and the creation of the data, then we classify the current contents of the shared object as *nonreproducible*. If the process is restarted from a checkpoint preceding the operation that is not reexecutable, then the resumed process may produce the shared object again with different contents than during the first execution. If the original contents of the object were communicated to another process, then the resumed process may produce data which is not consistent with the data that has already been sent to another process. Hence, the condition described above requires that we cause a checkpoint to occur whenever *nonreproducible data is sent to another process*. With this requirement, we know that the nonreproducible data will be recreated exactly

from the checkpoint information if the process crashes. Conversely, if we communicate *reproducible data* to another process, we do not need to do a checkpoint. If the process crashes, the data will either be restored from the last checkpoint or it will be recreated with exactly the same contents as before when the process reexecutes from the last checkpoint state.

An important consequence of the above discussion is that an application that operates mainly with values will have few operations that are not reexecutable. The application will therefore mainly create reproducible data and processes will not usually have to checkpoint when sending data. In other distributed shared memory systems, any access to shared data is not a reexecutable operation, since the shared data may be modified by another process between the time of the access by the original process and the access by the restarted process. Any data that is modified is therefore not reproducible, and processes must checkpoint any time they transmit data that they modified to another process.

4.2 Information Preserved By a Checkpoint

The basic state of any process on a host may be divided into the shared data managed by the SAM system and the processor's private state. We can minimize the amount of state that must be preserved at each checkpoint by recognizing which data on a processor will be available from another process or can be reconstructed during a recovery.

An important observation about the shared data is that only one copy of each data object need be preserved by checkpointing, even though several hosts may be caching copies of the object. For simplicity, we therefore designate a *main copy* of each data object and only checkpoint the main copy of each object. If a process using a cached copy (other than the main copy) fails, then the process holding the main copy can send a copy of the object to the recovering process. (Section 4.3 describes our method of ensuring that the main copy is not freed while it might still be needed for a recovery.) In our implementation, the main copy of a value is the copy on the process that created the value. The main copy of an accumulator is the actual current contents of the accumulator, which migrates between processes. We define the *owner* of an accumulator or value as the process that currently holds the main copy.

The *private state* of a processor consists of the following:

- processor state (mainly registers)
- stacks of all active tasks in the process

- values of statically allocated global variables
- shared data objects in the process of being created or modified
- pending requests by the current process for data owned by other processes
- pending requests by other processes for data owned by the local process
- directory information about the location of shared objects owned by other processes

The first four items are the local state of the parallel application, and the remaining items are essentially the private state maintained by the SAM system on each processor. Each processor must save its private state at every checkpoint. However, pending requests for data from other processes need not be saved during a checkpoint, since the other processes can reissue the requests during the recovery process. Also, directory information on the location of shared objects owned by other processes does not have to be saved, since the owners of those objects can transmit that directory information during a recovery.

Instead of using checkpointing or logging to disk to preserve the state of a process, we achieve fault tolerance by ensuring the state is replicated in two or more processes. We accomplish the replication of the shared data objects using the caching mechanism already provided by SAM. The replicated data is therefore available for use by the local process, potentially offsetting some the cost of the fault tolerance scheme. We will refer to a cached copy of an object used for checkpointing purposes as a *checkpoint copy*. We also replicate the private state of a process to another process as well. We therefore entirely avoid expensive disk operations during checkpointing. We also avoid depending on the local disk of a host for holding checkpoint information and can restart a process on a new host when the old host has a permanent failure.

Much of the shared data owned by a process is likely not to change at every checkpoint. We therefore keep track of which accumulators or values have been created and which accumulators have been modified since the last checkpoint and only replicate those objects at the next checkpoint. Objects that have not changed have already been replicated in a previous checkpoint. We currently preserve the entire contents of the private state at every checkpoint; we could instead attempt to save only the incremental changes to the private state since the last checkpoint.

To guarantee that we can recover from the simultaneous or near-simultaneous failure of n hosts, we must ensure that the private state and shared data owned by a

host is replicated at a minimum of n other hosts. In practice, because simultaneous failures are rare, we choose n to be 1, thereby minimizing the fault-tolerance overhead, but guaranteeing recovery only when one host failure occurs at a time. However, all of our techniques immediately generalize to the case where n is greater than one. To simplify the recovery procedure, we always replicate a particular object to a specific process which is determined directly from the name of the object. Similarly, we always replicate a process's private state to a specific process.

4.3 Memory Management of Replicated Data

Because the main copy and checkpoint copy of a data object are potentially needed to aid in a recovery, memory management of these copies must be handled specially. Normally, the SAM implementation can free the main copy when it determines (via user-provided information) that all accesses to the data have occurred. However, to allow for recovery from a failure, the SAM implementation must maintain the main copy somewhat longer, until every other process has done a checkpoint since its last access to that object. The reason is that some process which has accessed the object since its last checkpoint may crash. When the process is restarted from that checkpoint, the process will attempt to access the object. If the main copy has been freed up, then that access (hence the recovery procedure) will fail. Because the checkpoint copy is essentially a backup for the main copy (in case the process with the main copy is the one that fails), it can only be freed when the main copy is finally freed. These conditions on freeing the main and checkpoint copy are analogous to the conditions in other fault-tolerance methods on when checkpoint files and log records can be reclaimed.

We wish to avoid having to send any extra messages in order to determine when the main copy can be freed. Our approach is to mark the main copy as "freeable" at the point at which all accesses to it have occurred, but leave it in the shared object cache. At some point later, it will be replaced in the cache, because it is no longer being referenced. At that time, before actually freeing it, we must ensure that all processes have done a checkpoint since the copy was made freeable. The straightforward way of ensuring this fact is to send a message to each process and wait for a reply message indicating that it has done a checkpoint before freeing the copy. However, we have designed a technique which avoids these extra messages in almost all cases by piggybacking timestamp information on other fault-tolerance messages (message sent during checkpointing). The basic idea is to maintain information,

in all processes, on the last known time at which each process checkpointed, and to keep this information updated by transmitting with each fault-tolerance message the time when the sender last checkpointed. However, because of the lack of a globally consistent time among the hosts, the actual method is more complex.

Our technique is as follows. Each process i maintains a “virtual time” counter which is incremented each time there is a checkpoint or a freeing of an object that it owns. Process i also maintains a time vector T_i of the last-known virtual times on each process. The i_{th} component of T_i is always the current virtual time on process i . The rest of the vector is kept relatively current by transmitting the sender’s current time vector with each fault-tolerance message and using it to update the receiver’s time vector. Each process i also maintains vectors $C_i = (c_{i1}, c_{i2}, \dots, c_{i,n-1})$ and $D_i = (d_{i1}, d_{i2}, \dots, d_{i,n-1})$. C_i is the value of the time vector T_i when the process last did a checkpoint. d_{ij} is the last known value of c_{ji} from process j and is maintained by transmitting the current value of c_{ji} with each fault-tolerance message from process j to process i . If the current value of c_{ji} is c , then process j has done a checkpoint since the time on process i was c . Therefore, if the current value of d_{ij} on process i is d , then process j has done a checkpoint since the time on processor i was d . We can thus use the D_i vector to determine if each process has checkpointed since a particular virtual time on process i .

Suppose that we are about to remove the main copy of an object from the cache, and it was first marked freeable at time f on process i . If all elements of D_i are greater than f , we can free the copy immediately. If not, we must delay freeing the copy and send a “force-checkpoint” message to each process j for which $d_{ij} < f$. If process j receives a force-checkpoint message and its value of c_{ji} is less than f , then it does a checkpoint, which ensures that c_{ji} becomes greater than or equal to f . If its value of c_{ji} is already greater or equal to f , then process j does not need to checkpoint. In either case, process j replies to process i with its current values of T_j and c_{ji} . Process i can free the main copy when it receives replies from all the processes to which it sent force-checkpoint messages. It also informs the process holding the checkpoint copy that the checkpoint copy can be freed. We will present statistics in Section 5 showing that these force-checkpoint messages and forced checkpoints are required very infrequently in our applications.

4.4 Checkpointing Procedure

We can now describe the process of checkpointing in detail. Checkpointing is a transaction in which we

atomically save the private state of the process, replicate all the necessary data, and execute the operation that originally started the checkpoint. The transaction is accomplished by sending checkpoint copies to other processes initially in an “inactive” state, which makes the data inaccessible until the checkpoint commits. The steps of a checkpoint are as follows:

- Send a copy of the private state to a designated process.
- For each nonreproducible data object² owned by the current process which has not yet been checkpointed, send a checkpoint copy to a designated process in an “inactive” state.
- For each reproducible object owned by the current process which has not yet been checkpointed, send a checkpoint copy to a designated process (in an active state).
- Send the requested object (that caused the checkpoint) to the requesting process in an inactive state if it has not yet been sent to that process as a checkpoint copy.
- Wait for acknowledgements from the processes that received the private state or any inactive objects. Then “activate” all of the inactive objects by sending out a message to all of the processes with inactive objects.

While the local process is doing a checkpoint, it continues serving incoming requests for reproducible data, receiving data it has requested from other processes, and receiving replicated data sent by another process that is checkpointing. However, to ensure that each checkpoint is consistent, a process delays serving requests for nonreproducible data and responding to “activate messages” from other processes that have just completed a checkpoint. After the checkpoint has been committed, all of these delayed operations are executed, possibly causing another checkpoint in the process.

4.5 Recovering from a Failure

The goal of the recovery procedure is to restore the necessary state of the failed process for the SAM application to continue normally and to restore the necessary checkpoint information for the system to tolerate further failures. The state that must be restored includes

²In our current implementation, we do not detect operations other than access to accumulators that are not reexecutable. However, we could easily modify system libraries to catch the other kinds of operations, such as system calls, that are not reexecutable. None of the applications described in Section 5 perform these kinds of operations except for reading input files at startup.

all of the private state of the process, and all shared objects that were in use at the last checkpoint or whose main copy was on the failed process. The checkpoint information that must be restored are all the checkpoint copies that were maintained on the failed process.

Our system depends on the underlying message-passing layer to detect the failure of a process or host. Our current implementation runs on PVM3, which provides a notification message to surviving processes when another process fails. The recovery procedure when process p fails is as follows:

- One or more processes get a notification that process p failed. They send a message to a distinguished process d (process 0, or process 1 if process 0 is the process that failed) to coordinate the recovery.
- Process d receives the failure message and starts the recovery process, ignoring further messages indicating that process p has failed.
- Process d chooses a host on which to restart the failed process. This can be the same host, if it is still running or has been restarted, or a completely different host. Process d starts up a process in recovery mode on the new host.
- Process d sends a message to all other processes indicating that a recovery is in progress and including the PVM id of the new process. The new process will not receive any messages sent to the old process, since it has a different PVM id.
- The process holding the copy of the private state of the failed process sends that state to the new process.
- Each process besides the failed process aborts and restarts any checkpoint it has started that involves process p (since process p will have lost all information about that partial checkpoint).
- Each process besides the failed process also sends copies of data to the new process as follows:
 - If a process has a checkpoint copy of a data object whose main copy was at process p , it sends a copy of the data to process p , which will again hold the main copy.
 - If a process holds the main copy of a data object that was being accessed at the time of the last checkpoint by process p , it sends a copy of the data to process p .
 - If a process holds the main copy of an object whose checkpoint copy was on process p , it sends a checkpoint copy to process p .

– If a process holds the main copy of a data item and the directory information for that data was on process p , the process informs process p that it holds the main copy.

- Each process reissues any requests for data that have been made to the failed process and have not yet been fulfilled.
- When the recovering process has received all of the data from other processes, it resumes its computation with its restored registers, stacks, etc.

5 Performance Results

We have implemented our fault tolerance method for SAM applications running on clusters of workstations using PVM3. When linked with our fault-tolerant SAM library, a parallel SAM application recovers transparently when we kill one of the processes involved in the computation. In this section we give performance results for three long-running applications—GPS, WaTer, and Barnes-Hut—executing on a cluster of eight 225 MHz Alpha workstations connected by an ATM network. The workstations are connected by a 155 Mbit/sec AN2 ATM network [1, 19] developed at DEC SRC, and use a version of PVM3 that is highly tuned for the AN2 network. The maximum achievable bandwidth under PVM3 is about 14.6 Mbytes/sec and the latency for sending a message from one host to another is about 90 μ s. The timings for different runs of the applications are very consistent; the numbers below are from averaging the results for several runs, excluding the initialization phases, when there were no other users of the cluster. For each of the applications below, recovery from a failed process only takes on the order of a few seconds.

GPS is an application which attempts to determine a useful formula that predicts the degree of exposure to solvent of amino acids via a technique called genetic programming [7]. Genetic programming applications attempt to evolve useful formulas by emulating evolution and typically require large amounts of computer time. They build an initial population of individuals, which are candidate formulas, and then evolve the population over many generations using techniques analogous to genetic recombination, mutation, and survival of the fittest.

Figure 3 gives the speedup curves for the evolution of a population of 1000 individuals, both when not using fault tolerance (labeled GPS) and when using fault tolerance (labeled GPS-FT). The first three rows in the table to the right give the percent change in performance when fault tolerance is used in the runs, the average

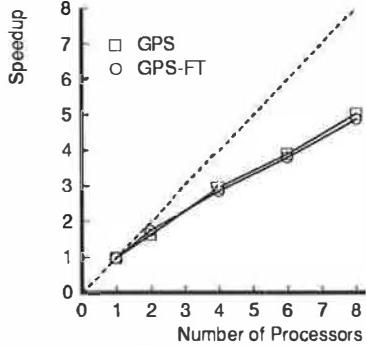


Figure 3: Performance and Fault-Tolerance Statistics for GPS

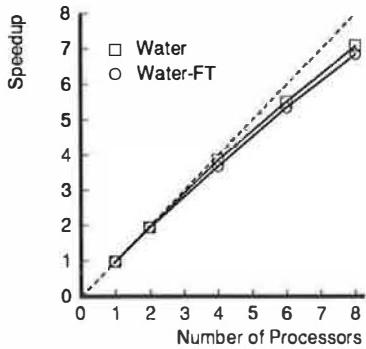


Figure 4: Performance and Fault-Tolerance Statistics for Water

number of checkpoints executed on each processor per second, and the percentage of sends of shared objects that cause checkpoints (equivalently, the percentage of sends of nonreproducible data). The low percentages indicate that many checkpoints are avoided and that the fault-tolerance overhead is much lower than it would be for systems which must assume that all data accesses are not reexecutable. The next two rows give the average number of “force-checkpoint” messages sent out on each processor per second and the average number of forced checkpoints on each processor per second. These operations, which are explained in Section 4.3, are never required for GPS runs. The final row gives the average miss rate on shared data both without and with fault tolerance. The miss rate is actually lower with fault tolerance because the replication of data during checkpoints can eliminate later misses to that data.

GPS is a relatively coarse-grained application, since there is much computation per individual in determining its fitness. In addition, the individuals of the population are evenly distributed across the processes, so the overhead of checkpointing the individuals is distributed well across processes. The fault-tolerance overhead is therefore quite low. GPS actually runs slightly faster on two hosts when using fault tolerance, because all the individuals created by one host are replicated to the

other host and hence are immediately available when that host wants them.

The Water application evaluates forces and potentials in a system of water molecules in the liquid state. Water is derived from the Perfect Club benchmark MDG [3] and performs the same computation. Water is implemented using Jade [14], which is a parallel language implemented entirely in SAM. Since we have added fault tolerance to the SAM system, all Jade applications are also automatically fault-tolerant.

Figure 4 gives speedup curves for the simulation of 1728 particles, both when not using fault tolerance and when using fault tolerance, and with the same statistics as before. In Water, the main process collects all the data at each time step, so it must checkpoint most of the shared state of the system, which is quite large. The main process can therefore become a bottleneck with increasing processors, but the actual overheads remain quite small. Because of Jade’s load-balancing functionality, the distribution of tasks to processors involves an operation which is not reexecutable on the receiving process. Since tasks cause checkpoints only upon completion when they communicate their results, all data produced by these tasks is considered non-reproducible. Some force-checkpoint messages and forced checkpoints occur in Water, but the number is

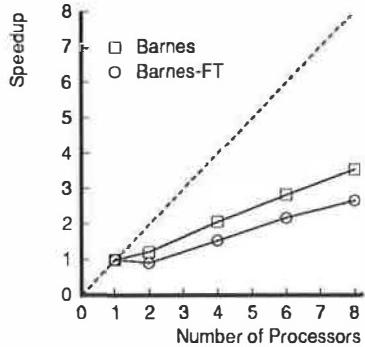


Figure 5: Performance and Fault-Tolerance Statistics for Barnes-Hut

very small. Again, the miss rate is actually lower with fault tolerance because the replication of objects during checkpoints eliminates some misses to these objects.

Barnes-Hut [2] is an application that simulates the evolution of an n -body system using a tree data structure to compute the forces between bodies in $O(n \log n)$ time. The SAM version of the code is based on the original serial version of the code and is written in a shared-memory style. All of the processes participate in the building and processing of the distributed tree data structure, and communication between processes is much more fine-grain than in GPS and Water.³ The force computation is partitioned across processes so that each process has extensive locality in accessing the distributed tree, which is exploited by the dynamic caching provided by SAM.

Figure 5 gives speedup curves for a simulation of 8000 bodies. There is significant software overhead in implementing the Barnes-Hut algorithm in a shared-memory style on distributed memory machines, but this overhead only reduces the overall performance by a fixed percentage and performance scales well. Because of the fine-grain nature of the application and the large tree data structure that is built, the fault-tolerance overhead is quite high. The communication in building and modifying the tree causes many checkpoints, and the checkpointing process adds significantly to the latency of communication. If the Barnes-Hut application did not have strong locality which reduces the amount of fetching of remote data, the number of checkpoints would be even higher. The number of checkpoints is also greatly reduced by the high percentage of sends of reproducible data. Two conflicting factors affect the shared data miss rate when fault tolerance is used. As described above, the miss rate can be reduced if processors which have received checkpoint copies of an object then access that object. However, because of

	2 proc	4 proc	6 proc	8 proc
% change in performance	-25.0	-25.9	-22.4	-24.6
checkpoints per second	0.66	1.44	1.40	2.24
% sends causing checkpoints	16.3	6.8	5.9	5.3
force-ckpt messages per sec.	0.38	1.55	0.00	0.00
forced checkpoints per sec.	0.09	0.08	0.00	0.00
shared data miss rate (%)				
without fault tolerance	0.051	0.132	0.186	0.305
with fault tolerance	0.013	0.123	0.193	0.267

the extra checkpoint copies, the shared data cache may be utilized less efficiently for other objects. Because of these conflicting effects, the shared data miss rate decreases for some runs when fault tolerance is used, but increases for others.

Our performance numbers show that our fault-tolerance method has low overhead and is quite suitable for the coarse-grain applications that would typically be run on networks of workstations. The overhead of our method is larger for the finer-grain Barnes-Hut application, but the parallel performance still scales well despite the overhead.

6 Related Work

In this section, we compare with recent work that have provided fault tolerance in the context of software distributed shared memory.

Like our work, Kaashoek et al. [10] implement fault tolerance for a shared object system called Orca. However, the methods for achieving fault tolerance are completely different. Kaashoek et al. use global checkpoints at periodic intervals to ensure that the system can be restarted from a previous state if there is a host failure. Checkpointing the state of the entire system consistently is fairly simple, because their implementation of Orca is based on reliable ordered broadcasting of all messages. Like our system, they provide fault tolerance transparently without any extra effort by the Orca programmer. Because they use global checkpointing, they can recover from failures of many hosts simultaneously (assuming that none of the disks used for checkpointing have permanent failures). However, their fault tolerance method depends completely on their use of reliable ordered broadcast, whose performance does not scale well for large numbers of hosts or networks with variable delays.

MOM [5] is a fault-tolerant implementation of a Linda-like programming model. The programmer uses

³The average time between shared data misses on each process in 8-processor non-fault-tolerant runs is 109 ms for GPS, 484 ms for Water, and 5 ms for Barnes-Hut.

the basic operations of Linda, but must sometimes specify additional information that is necessary only to support fault tolerance. In addition, the programmer is constrained to a model of parallel computation in which each task reads some input tuples and produces some output tuples, but otherwise does not interact with other tasks. Xu and Liskov [21] describe a design for a distributed implementation of Linda in which the tuple space is made highly available by replicating it across several processors. However, they do not consider the problem of restarting Linda tasks that were running on hosts that have failed.

Wu and Fuchs [20] describe techniques for making a page-based shared virtual memory system fault-tolerant. Like our work, Wu and Fuchs implement asynchronous pessimistic checkpointing by forcing a checkpoint whenever a host sends a page to another host in response to a request. They propose using two copies of each virtual memory page to allow checkpointing of pages to disk to proceed quickly. Janssens and Fuchs [8] apply this method to a DSM system with a relaxed memory consistency model. The expected checkpointing overhead is greatly reduced because a host need only checkpoint when it holds a synchronization variable that another host attempts to acquire. Richard and Singhal [13] describe a related approach in which a processor logs to disk the pages that it has accessed via read operations whenever it sends a modified page to another processor, instead of doing a full checkpoint. A processor must periodically checkpoint (independently of other processors) in order to reduce the size of the log.

Neves et al. [12] describe a method of providing fault-tolerance for an entry-consistent DSM system called DiSOM. Each process logs the contents of an object in its own local volatile memory, whenever it releases a write lock on the object. If another process acquires the object and later fails, then the data can be recovered from the process that holds the log entry. This technique is similar to our approach of not freeing data objects until we can guarantee that all other processes have checkpointed since their last access to the data. Each process must checkpoint to disk frequently to reclaim the memory used by the log entries.

Of the methods described above, only those associated with Orca and MOM have been implemented.

7 Conclusion

We have described a method of providing transparent fault tolerance for long-running parallel applications on networks of workstations in the context of the SAM shared object system. Our fundamental approach is

to use the dynamic caching functionality provided by SAM to ensure the replication of all data on more than one host at checkpoints. Our method thus avoids expensive writes to disk and does not require a common file server. Each process checkpoints independently and only when sending data which is not reproducible to another process. The SAM design, which provides extra information on how data is accessed, is quite useful in reducing the number of checkpoints.

Our method works on heterogeneous clusters of workstations (which have a common message-passing layer such as PVM3), without changes to the system software and without doing any global synchronizations among the hosts. When there is a failure, only the work of the failed process must be redone from the last checkpoint; other processes continue without rollback. Interestingly, in a task-based application, checkpoints naturally occur at task boundaries when the results of tasks are transferred between processes, thus ensuring that only the work of the currently executing task must be redone during a recovery. We are investigating further optimizations to reduce the checkpoint overhead, such as using a more sophisticated scheme for determining where to replicate data or using available information to decrease the amount of state saved during a checkpoint.

It is interesting to note that our method is potentially useful for fast process migration. We can move a process in a parallel application off a particular host very quickly by killing it and restarting it on another host from its last checkpoint. Such fast process migration is useful when a parallel application is running on idle workstations that are on individuals' desks and an owner begins to use one of the workstations again.

We have implemented our method in SAM for workstation clusters using PVM3. Our method successfully recovers from process and host failures during long-running SAM applications. The amount of overhead is dependent on the communication patterns of an application and may be significant for applications with finer-grain communication such as Barnes-Hut. For the coarse-grain applications that are more likely to be run on networks of workstations, the overhead is quite low.

Acknowledgments

We thank Chandu Thekkath, Mike Burrows, and DEC SRC for allowing us to use the AN2 cluster and for providing a fast version of PVM3 for the cluster. We thank Simon Handley for the use of his GPS code and Martin Rinard for providing the Water code. We also thank the referees for comments that helped improve the paper.

Availability: The SAM system is available via anonymous FTP at suif.stanford.edu in /pub/sam and via the World Wide Web at <http://suif.stanford.edu>.

References

- [1] T. E. Anderson, S. S. Owicki, J. B. Saxe, and C. P. Thacker. High-speed switch scheduling for local-area networks. *ACM Trans. Comput. Syst.*, 11(4):319–352, Nov. 1993.
- [2] J. E. Barnes and P. Hut. A Hierarchical $O(N \log N)$ Force-Calculation Algorithm. *Nature*, 324(6096):446–449, Dec. 1986.
- [3] M. Berry and et al. The Perfect Club Benchmarks: Effective Performance Evaluation of Supercomputers. *International Journal of Supercomputer Applications*, 3(3):5–40, 1989.
- [4] A. Borg, J. Baumbach, and S. Glazer. A Message Passing System Supporting Fault-Tolerance. In *Proceedings of the ACM SIGOPS Symposium on Operating System Principles*, pages 90–99, Oct. 1993.
- [5] S. R. Cannon. Adding Fault-tolerant Transaction Processing to Linda. *Software – Practice and Experience*, 24(5):449–466, May 1994.
- [6] K. M. Chandy and L. Lamport. Determining Global States of Distributed Systems. *ACM Transactions on Computer Systems*, 3(1):63–75, Feb. 1985.
- [7] S. Handley. The Prediction of the Degree of Exposure to Solvent of Amino Acid Residues via Genetic Programming. In *Second International Conference on Intelligent Systems for Molecular Biology*, pages 156–160, Stanford, CA, 1994.
- [8] B. Janssens and W. K. Fuchs. Relaxing Consistency in Recoverable Distributed Shared Memory. In *Proceedings of the Twenty-third International Symposium on Fault-Tolerant Computing*, pages 155–163, July 1993.
- [9] D. B. Johnson and W. Zwaenepoel. Recovery in Distributed Systems Using Optimistic Message Logging and Checkpointing. In *Proceedings of the Seventh Annual ACM Symposium on Principles of Distributed Computing*, pages 171–181, Aug. 1988.
- [10] M. F. Kaashoek, R. Michiels, H. E. Bal, and A. S. Tanenbaum. Transparent Fault-Tolerance in Parallel Orca Programs. In *Symposium on Experiences with Distributed and Multiprocessor Systems*, pages 297–311, Mar. 1992.
- [11] K. Li, J. F. Naughton, and J. S. Plank. Checkpointing Multicomputer Applications. In *Proceedings of the Tenth Symposium on Reliable Distributed Systems*, pages 2–11, Sept. 1991.
- [12] N. Neves, M. Castro, and P. Guedes. A Checkpoint Protocol for an Entry Consistent Shared Memory System. In *Proceedings of the Thirteenth ACM Symposium on Principles of Distributed Computing*, pages 121–129, Aug. 1994.
- [13] G. G. Richard III and M. Singhal. Using Logging and Asynchronous Checkpointing to Implement Recoverable Distributed Shared Memory. In *Proceedings of the Twelfth Symposium on Reliable Distributed Systems*, pages 58–67, 1993.
- [14] M. C. Rinard, D. J. Scales, and M. S. Lam. Heterogeneous Parallel Programming in Jade. In *Proceedings of Supercomputing '92*, pages 245–256, Nov. 1992.
- [15] D. J. Scales and M. S. Lam. An Efficient Shared Memory System for Distributed Memory Machines. Technical Report CSL-TR-94-627, Computer Systems Laboratory, Stanford University, July 1994.
- [16] D. J. Scales and M. S. Lam. The Design and Evaluation of a Shared Object System for Distributed Memory Machines. In *Proceedings of the First Symposium on Operating System Design and Implementation*, pages 101–114, Nov. 1994.
- [17] R. E. Strom and S. Yemini. Optimistic Recovery in Distributed Systems. *ACM Transactions on Computer Systems*, pages 204–226, Aug. 1985.
- [18] V. Sunderam. PVM: a Framework for Parallel Distributed Computing. *Concurrency: Practice and Experience*, 2(4):315–339, Dec. 1990.
- [19] C. P. Thacker and M. D. Schroeder. AN2 Switch Overview. In preparation.
- [20] K.-L. Wu and W. K. Fuchs. Recoverable Distributed Shared Virtual Memory. *IEEE Transactions on Computers*, 39(4):460–469, Apr. 1990.
- [21] A. S. Xu and B. Liskov. A Design for a Fault-tolerant, Distributed Implementation of Linda. In *The Nineteenth International Symposium on Fault-Tolerant Computing*, pages 199–206, June 1989.

Author Information

Dan Scales is a researcher at Digital Equipment Corporation's Western Research Laboratory. His research interests include parallel languages and runtime systems, distributed systems, and software development environments. He received a BS in electrical engineering and computer science from Princeton University and a PhD in computer science from Stanford University in 1995. He may be contacted at scales@pa.dec.com.

Monica Lam is an associate professor in the Computer Science Department at Stanford University. Her research interests are in compilers, computer architectures, and parallel computing, and she currently leads the SUIF (Stanford University Intermediate Format) parallelizing compiler research project at Stanford. She received a BS degree from the University of British Columbia and a PhD in computer science from Carnegie Mellon University in 1987. She is the recipient of a National Science Foundation 1992 Young Investigator Award. She may be contacted at lam@cs.stanford.edu.

Why Use a Fishing Line When You Have a Net?

An Adaptive Multicast Data Distribution Protocol

Jeremy R. Cooperstock and Steve Kotsopoulos

*Department of Electrical and Computer Engineering
University of Toronto*

Abstract

The design and implementation of a system to provide reliable and efficient distribution of large quantities of data to many hosts on a local area network or internetwork is described. By exploiting the one-to-many transmission capabilities of multicast and broadcast, it is possible to transmit data to multiple hosts simultaneously, using less bandwidth and thus obtaining greater efficiency than repeated unicasting. Although performance measurements indicate the superiority of multicast, we dynamically select from available transmission modes so as to maximize efficiency and throughput while providing reliable delivery of data to all hosts. Our results demonstrate that file-distribution programs based on our protocol can benefit from a substantial speed-up over TCP-based programs such as `rdist`. For example, our system has been used to distribute a 133 Kbyte password file to 68 hosts in 20 seconds, whereas the equivalent `rdist` took 251 seconds.

1 Introduction

Distributing data to multiple hosts using connection-oriented protocols such as TCP [1] can be inefficient because the data must be transmitted over the network multiple times, once to each target. Popular file distribution programs, including `rcp`, `rdist`, and `track` [2], are all based on this protocol. The time they require to distribute files to a group of machines is proportional to the number of machines in that group.¹

¹Recent improvements to `rdist` include provisions for parallel update [3], which involves sending files to a number of clients simultaneously. Using this method, file distribution time can be reduced by a factor of almost four. However, repeated unicasting is still used as the transmission mode, and thus, distribution time remains dependent on group size.

Modern local area networks, such as Ethernet and FDDI, support one-to-many communication via broadcasting [4] and multicasting [5]. By exploiting these capabilities, it is possible for a system to send data to multiple hosts simultaneously, thereby greatly reducing network traffic, host load, and elapsed time. Since broadcast and multicast packets can be sent only as datagrams, it is necessary to forgo connection-oriented protocols and instead implement the system using connectionless protocols such as UDP [6].

Unfortunately, the UDP protocol is unreliable: neither delivery nor ordering of UDP data packets is guaranteed. A further problem is that broadcast packets cannot travel outside of a local area network, and while multicast does not suffer this limitation, it is not supported by all hosts. To address these problems, we designed and implemented the *Adaptive File Distribution Protocol* (AFDP). AFDP provides reliable, rate-controlled delivery of data to multiple hosts, automatically determining the *best* transmission mode, according to the number of hosts, their capabilities, and support from the connecting networks. The protocol is implemented in two co-operating programs, similar in function to `rcp`, but capitalizing on the connectionless facilities of UDP. With its additional tuning features, AFDP allows the user control over the trade-off between speed (throughput) and host and network loading.

Using the AFDP programs as building blocks, we have constructed a prototype file update application, called `afdpdist`. Although it is commonly held that a performance penalty must be paid for a large group, our experimental results indicate that `afdpdist` can distribute files to a group of machines in time independent of group size. While `afdpdist` is rather crude, we hope that it can be used as a proof-of-concept to design and build a better `rdist`.

We now survey related work to AFDP, then describe the AFDP protocol. Following this, we discuss the performance of our system and summarize some applications of AFDP including *afdpdist*.

2 Related Work

The most relevant related work is that of the *Reliable Multicast Protocol* (RMP) [7], recently implemented by Whetten et al. Similar to the MBusI [8] and the Totem protocol [9], it provides reliable, ordered delivery of data. Like AFDP, RMP goes beyond these earlier systems by allowing multiple, simultaneous senders, and does not rely on hosts to provide multicast support.

Similar systems have been proposed and implemented, but these generally rely on a single transmission mechanism. For instance, Ioannidis et al. implemented CFDP [10], a one-to-many distribution system without any flow control facilities. Oki et al. implemented *The Information Bus* [11], which uses a retransmission protocol to provide reliable delivery semantics. Because these systems implement their group communications using broadcast, exclusively, they cannot be used to distribute data beyond a LAN.

Clark et al. describe the NETBLT protocol [12] for rapid transfer of large quantities of data between computers. To achieve high throughput, NETBLT uses a transmission rate-control algorithm similar to ours. However, since this is a connection-oriented protocol, it is not applicable to efficient group communications.

Other research has dealt with multicast transport protocols and flow control problems. For example, Cheriton describes the *V Distributed System* [13], which also utilizes multicast communication primitives, but only offers “best-effort,” not reliable delivery. Armstrong et al. propose a *Reliable Multicast Transport* [14] protocol to provide a network service that could be used to implement a system such as ours. Unlike AFDP, though, it cannot be supported by hosts that do not have multicast capability.

Birman et al. constructed the ISIS system [15, 16], which offers reliable broadcast and multicast as part of its toolkit approach to distributed systems design. Unfortunately, ISIS requires separate acknowledgements from each destination, which limits performance and scalability.

3 The Adaptive File Distribution Protocol

AFDP was designed with the goals of efficiency and flexibility. We wanted to distribute large files², typical of operating system upgrades and the increasingly common image and MPEG files, to all group members reliably, efficiently, and as quickly as possible. To provide maximum flexibility, multiple hosts should be able to transmit data concurrently. The network, as well as all participating hosts, are assumed to support (unreliable) broadcast transmissions, and may also support multicast. It is also assumed that varying network and host loads will cause fluctuations in available network bandwidth as well as reception capabilities of the hosts.

These requirements motivated an approach that capitalizes on available communication modes and provides a low-overhead, rate-based flow-control strategy to ensure reliable delivery. Note that we presently provide very little fault tolerance. If a network partition arises between group members, or a host fails, we make no attempt to recover from the error. This is an area for future work.

3.1 Flow Control

Traditional flow control in transport protocols including *stop-and-wait* and *go-back-N* were designed with point-to-point connections in mind [17]. Because these techniques require positive acknowledgements from hosts receiving data, their performance tends to suffer as the group size, and hence return traffic to the sender increases. The *selective-repeat* technique addresses this shortcoming with a negative acknowledgement scheme: the sender transmits the entire message, as a series of message packets, to a group of receivers that collate the packets by sequence number. If a receiver discovers that it is missing a packet, by detecting a gap in the sequence numbers, it asks the sender to retransmit it. However, this technique may suffer if the network is frequently dropping data, thus requiring the sender to retransmit many packets.

AFDP combines selective-repeat with a rate-based flow control strategy to prevent unacceptable packet losses. Like Henriksen [17], we wish to avoid wasting bandwidth by unnecessary retransmissions. Therefore, our system attempts to maintain a transmission rate that is as high as possible without resulting in dropped packets.

²The current range of sequence numbers allows AFDP to support up to 4 terabyte files.

3.2 Protocol Overview

AFDP is based on the publishing model [18], in which any host receiving data is called a *subscriber* while any host sending data is called a *publisher*. One special subscriber is designated as the *secretary* for each group; this host is responsible for managing group membership, authorizing publishers, and determining the appropriate transmission mode to be used. Publishing does not require explicit acknowledgements of received data. Instead, subscribers use negative acknowledgements as a means of requesting retransmission of missing or corrupted packets.

The interaction between subscribers, publishers, and secretary is depicted in Figure 1. Subscribers wishing to receive messages may join the group at any time by contacting the secretary. The identity of the secretary is either specified on the command line or found by broadcasting or multicasting a FIND_SECRETARY query to a well-known port. If no secretary replies, the subscriber may become the group secretary, thereby creating a new group. In the case of races, a voting procedure selects the secretary with the highest IP number. The secretary may disband the group at any time by sending a SHUTDOWN message to each subscriber.

Because the secretary is typically a long-lived process running on a reliable host, whereas subscribers may join and leave the group at any time, we separated the functionality of these two processes. The secretary maintains information about the group in the form of a host table. It listens on three ports for messages: one well-known port for FIND_SECRETARY queries, an *admin* port for JOIN and LEAVE requests from subscribers, and a *publish* port for PUBLISH.REQUEST messages from publishers.

JOIN and LEAVE requests may be issued by subscribers at any time. Each provides the identity of the subscriber, and the name of the group the subscriber wishes to join or leave. In addition, the JOIN request specifies whether or not the subscriber is capable of receiving multicast packets. The secretary replies to these requests with a JOIN.REPLY or a LEAVE.REPLY message, as appropriate.

While the secretary will allow hosts to join groups even while data is being published, these new subscribers may not influence the publishing rate until the start of the next message. If, in the event that data packets for the current message are being distributed in multicast or broadcast mode, all subscribers may receive these packets. To avoid

confusion, new subscribers simply ignore packets from messages that were begun earlier than their JOIN.REPLY timestamps.

When a publisher wishes to transmit a message, it first sends a PUBLISH.REQUEST to the secretary, who in turn will either refuse the request or grant permission. The secretary will only refuse a PUBLISH.REQUEST if the number of active publishers exceeds some threshold. We are presently adding features to prevent all but a select number of hosts from publishing. If a host is granted publishing permission, it is told which transmission mode to use, as well as an initial suggested inter-packet interval, T_p . If unicast will be employed, the secretary also provides the publisher with a current list of subscribers.

The secretary, in consultation with the publisher, is responsible for selecting a transmission mode that is appropriate for all subscribers. Since multicast is considerably more efficient than broadcast, this mode is chosen whenever all hosts support it. However, if some hosts do not support multicast, the secretary must choose between unicast and broadcast. (A planned enhancement to AFDP would allow the secretary to instruct the publisher to multicast to some hosts and unicast to others.) Since broadcast packets cannot travel beyond a LAN, this option is viable only when all subscribers are on the same LAN as the publisher. If so, the secretary will select unicast when the number of subscribers is below some *threshold*, and broadcast, otherwise. Note that when the group consists of two or more subscribers, unicast mode will require multiple transmissions of each packet. The threshold at which AFDP switches from unicast to broadcast mode can be controlled by the user, allowing one to influence the tradeoff between network loading and CPU usage.

Once granted permission, the publisher transmits the message as a sequence of data packets at intervals of T_p milliseconds. Each packet contains the current sequence number, as well as the total number of packets in the message. Under the negative acknowledgement scheme, subscribers are silent unless they require the publisher to resend one or more packets. Subscribers perform a sequence check function at intervals of a fixed wait time, T_w , to determine if any packets are outstanding. This will be true if there is a gap in received sequence numbers or if no packets have arrived since the last sequence check. Note that a packet is considered successfully received only if its data was not corrupted.

If any packets are outstanding, the subscriber

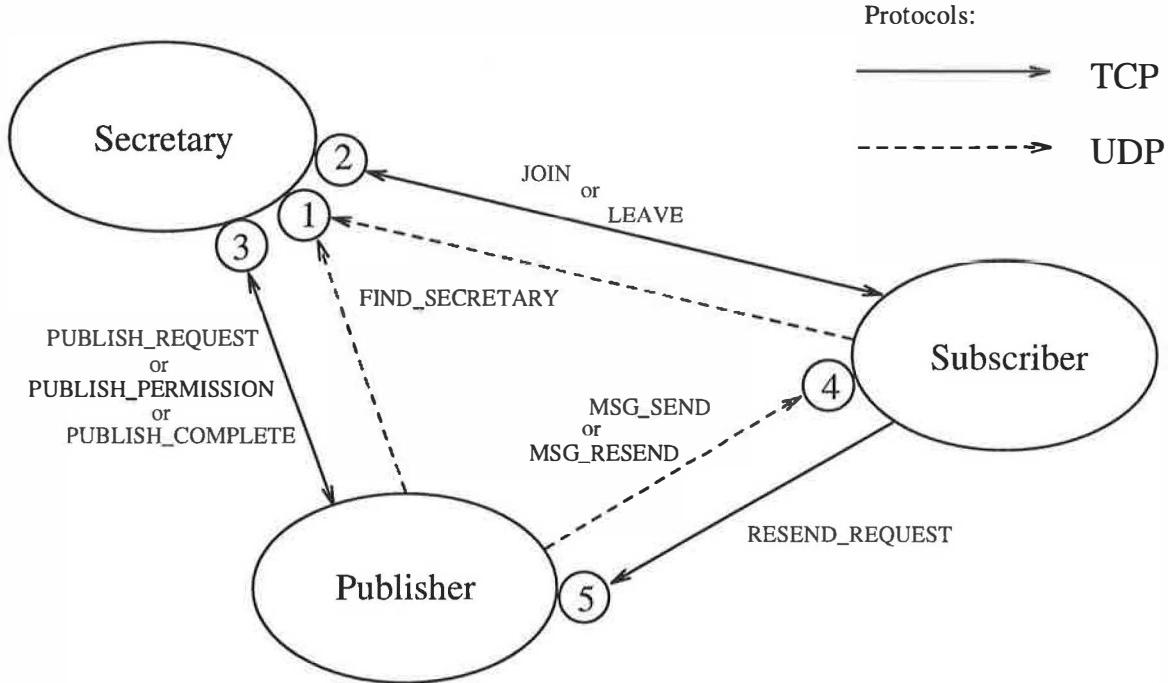


Figure 1: Interprocess communication in AFDP. The ports are defined as follows: 1. search port. 2. admin port. 3. publish port. 4. group data port. 5. control port. Note that a group may consist of multiple subscribers and publishers, although for simplicity, we only show one of each.

issues a RESEND_REQUEST, which is simply a negative acknowledgement of the missing packets, to the control port of the publisher. This contact is handled via TCP, which incurs a minor penalty for the connection set-up time, but ensures reliable delivery of the NAK. This is of paramount importance, especially in light of the role this message plays in forcing the publisher to slow down its transmission. Provided that the request is valid (i.e. the subscriber is an authorized group member and the packets it has requested have already been sent to the group), the publisher first increases the value of T_p , then services the retransmission request before returning to normal publishing. The more RESEND_REQUESTS serviced, the more the publisher must slow down.

This technique may present a danger in the case of temporary partitions, in which the publisher could potentially be flooded by an implosion of negative acknowledgements. To reduce the likelihood of this occurring, subscribers delay a random amount of time before requesting resends of missing packets. A RESEND_REQUEST is then issued only if some packets are still missing at the end of the delay. Furthermore, all retransmitted packets are sent to the entire group, based on the assumption that if one host has missed

a packet, it is likely that others have too. We are presently experimenting with additional techniques intended to reduce further the chance of multiple subscribers requesting a resend of the same packet. One such method is for the publisher to inform subscribers in advance which packets it is about to resend.

Similar to VMTP [19], the publisher periodically decreases the value of T_p to ensure that it is transmitting at the maximum rate the subscribers can handle. In order to prevent AFDP from overloading the network, we typically enforce a reasonable minimum on T_p . However, a *rabid* mode that can be used to override this minimum, as well as a *nice* mode that raises this minimum to 100 msec, are provided as options.

Under ordinary circumstances, our rate-based flow control mechanism minimizes unnecessary retransmissions. As soon as one subscriber notices that packets have been dropped, the publisher is forced to decrease its transmission rate, thus relieving pressure on the network. While this technique reduces publisher throughput, it keeps the number of retransmissions reasonably low.

packet size (bytes)	publisher throughput (Kbytes/s)		
	unicast	multicast	broadcast
9000	820	818	N/A
4096	405	407	N/A
2048	203	204	N/A
1472	143	142	143
1024	101	102	102

Table 1: Publisher throughput vs. packet size for different transmission modes. Note that throughput for multicast and broadcast modes is independent of group size, provided all hosts are approximately the same speed and under the same load.

3.3 Packet size

Table 1 demonstrates that using larger packets to transmit data provides greater throughput by reducing protocol overhead and thus increasing efficiency [19] [17]. Therefore, we are motivated to avoid the use of broadcast, as this transmission mode imposes a limit of 1472 bytes on packet size. This limit is due to the fact that broadcast packets cannot be fragmented by the IP layer, and so must fit into a single 1500 byte Ethernet frame [20]. Allowing for the UDP and IP headers, the maximum size of a broadcast UDP packet is 1472 bytes. However, on some MIPS machines, we found that the largest allowable broadcast UDP packet size was 1468 bytes, a value MIPS selected to work around a DMA problem, so we use this value for portability. Other architectures allow a broadcast packet size of 1472 bytes.

We observed that for the same packet size, throughput is relatively independent of transmission mode. As a result of the multicast data packet size limit being over six times that of broadcast packets, we obtain a corresponding improvement in peak throughput, as seen in table 1. Hence, we strongly favor multicast over broadcast whenever both are supported. Ioannidis' CFDP [10] uses a packet size of 512 bytes to avoid fragmentation but we did not observe any benefit in doing so.

3.4 Internetwork Extensions

There are two considerations in scaling this protocol to an internetwork environment. We note that multicast is not universally supported and that broadcast packets cannot travel beyond a LAN. To cope with these restrictions, we presently unicast to remote subscribers, unless all subscribers and intervening routers support multicast, in which case we use that transmission mode.

Where multicast is not supported, subscribers and publishers on remote LANs must specify the identity of the secretary host on the command line. This way, the exchange of information with the secretary can proceed with unicast packets.

4 Performance

The performance measurements of Figure 2 were made on the University of Toronto's Engineering Computing Facility (ECF), a heterogeneous workstation cluster consisting of one MIPS RC6280, eleven SGIs, ranging from R4000 Indys to R4400 multiprocessors, and 30 SUN Sparc10 systems. The machines are on a lightly-loaded Ethernet, connected to several other LANs. Unless noted otherwise, all of our measurements were performed using the default AFDP parameters.

4.1 Distribution Time

For each test, a 7.0 Mybyte file was transferred from the publisher to all subscribers. The peak publisher throughput observed for AFDP was between 800 and 900 Kbytes/s in unicast and multicast modes, and 140 Kbytes/s in broadcast mode. AFDP was also evaluated on various other clusters and across multiple clusters with similar results. When multicast is supported by all hosts and the intermediate networks, transfer time is determined by the network and the speed of the slowest machine, rather than the number of subscribers. As Cooperstock and Kotsopoulos discuss in a previous paper [21], even in the worst case, when unicast must be used to all hosts, performance is still approximately linear in group size.

As can be seen from Figure 2, the performance of AFDP is primarily determined by the communication mode used. In unicast mode, data must be sent to each subscriber separately, so the transfer time is proportional to the number of hosts. However, in broadcast or multicast mode, data is sent only once, provided no subscribers request a retransmission, and received by all subscribers, so the transfer time remains constant, independent of group size. Using broadcast, AFDP distributed the file in 57 seconds, while with multicast, the time was 15 seconds.

When one or more subscribers are not capable of receiving packets at the current publishing rate, they communicate this to the publisher through RESEND_REQUESTS. This means that the publishing rate can be dictated by the slowest subscriber in the group. While this may not be a

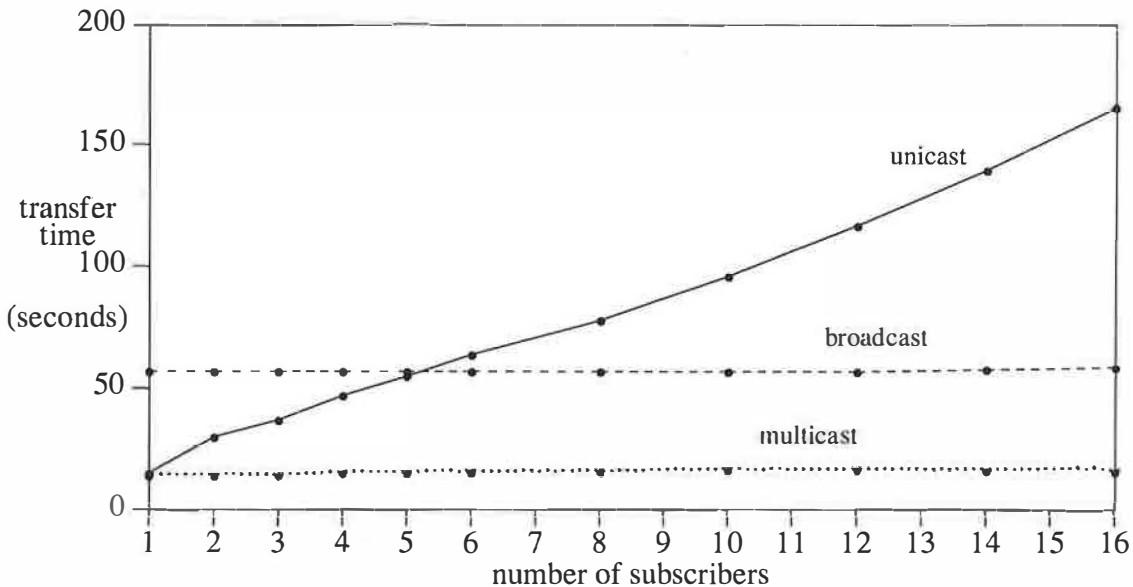


Figure 2: AFDP transfer time vs. number of subscribers using unicast, multicast, and broadcast modes. The transferred file was 7.0 Mbytes in each case. Note that because of the smaller packet size required by broadcast data, this mode takes approximately six times as long as multicast (or unicast to only one subscriber).

desirable characteristic for all applications, our focus on reliable file distribution required such an approach.

We measured peak *ftp* performance at approximately 900-1000 Kbyte/s between two SGI R4000 and between two Sun IPC machines, although typical average throughput may be substantially less. AFDP consistently matches this performance.³ on lightly loaded machines. While these numbers may seem unimpressive for our protocol, it must be noted that AFDP can maintain similar publisher throughput for many subscribers, provided that multicast is supported by all members. In this case, files may be distributed to an entire group as quickly as they could be to a single subscriber, whereas for any TCP-based protocol, the distribution time is proportional to group size. As pointed out by one reviewer, the linear relationship of distribution time to group size for TCP-based protocols could be beaten by arranging the receivers (subscribers) on multiple ethernet segments. However, we submit that such architectures are not always available.

4.2 Adaptivity

As shown in the saw-tooth graph of Figure 3, AFDP dynamically adapts to current network conditions as determined by subscriber feedback. While subscribers are receiving data packets correctly, the publisher slowly decreases the inter-packet delay, T_p , to a minimum of 10 ms. However, for every RESEND_REQUEST issued by a subscriber, the publisher immediately increases the inter-packet delay by 10 ms. This balances the need for quick adaptation to problems with a conservative attempt to maximize throughput.

4.3 Transmission Mode

Since broadcast packets are received by every host on the LAN, they can waste CPU cycles on non-participating systems. For example, on an SGI Indy R4000PC, the processing of broadcast packets accounted for approximately 11% of CPU usage. For this reason, we allow the user to select the threshold at which AFDP switches from unicast to broadcast, via a command-line argument. Sites that wish to avoid broadcasts can set this threshold as desired, to tailor the tradeoffs for their environment. Based on the transfer times of unicast and broadcast modes in the previous graph, it would appear logical to select a threshold value of six.

³On the Sun IPC machines, this figure is attained in *rabit* mode. In order to be "network-friendly," we normally avoid this mode.

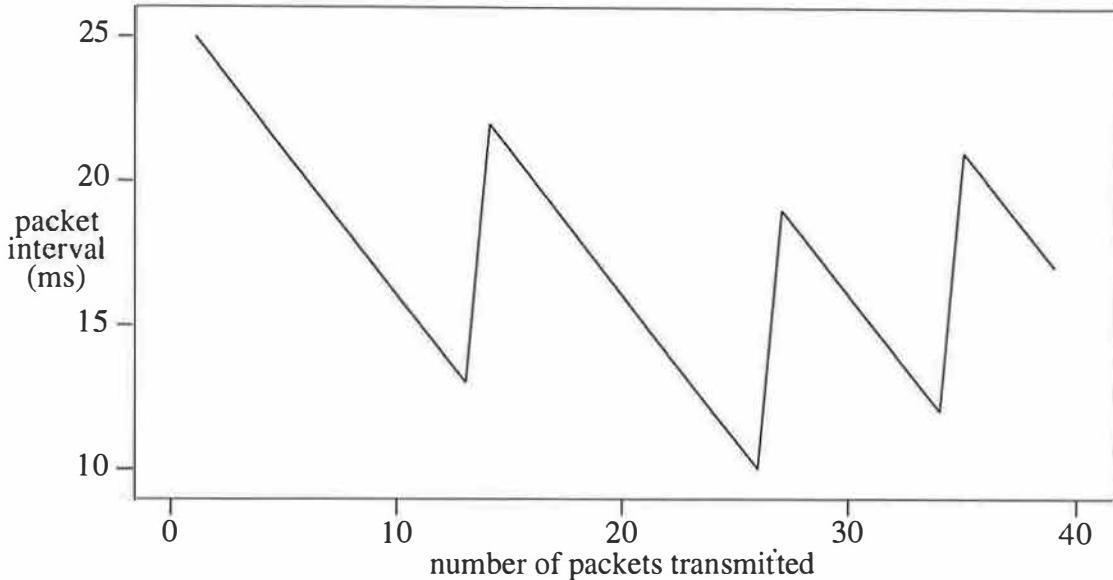


Figure 3: Inter-packet delay, T_p , as a function of packet number. Publishers decrease the delay by 1 ms/packet unless a subscriber issues a RESEND_REQUEST. In this case, the publisher increases the delay by 10 ms, resulting in the saw-tooth appearance of the graph.

network	ftp throughput (Kbytes/s)
idle	470
rcp	150
AFDP (broadcast)	430
AFDP (multicast)	200
AFDP (unicast)	200

Table 2: ftp throughput while rcp or AFDP is running. The AFDP transfer was to three subscribers in each case.

4.4 Network Load

While high throughput is attractive, AFDP, by default, does not attempt to consume all available network bandwidth. However, in the *rabid* mode, AFDP will attempt to utilize the full bandwidth available. To test the default mode, an ftp transfer between a MIPS 6280 and a Sparc 10 was run on the same network under the following conditions: relatively idle, with a large rcp transfer running, and finally, with a large AFDP transfer running in each of the three transmission modes. The results of Table 2 indicate that while AFDP impacts other users of the network, it is no worse in this regard than rcp, which, by contrast, consumes far more bandwidth.

The performance of our file update program, afpdist, discussed in further detail in Sec-

tion 5.1, was then compared with various other file update programs. For each program, we distributed a directory containing 7.0 Mbytes of files of various sizes to subscriber hosts. The results, shown in Figure 4, demonstrate that afpdist offers a linear speedup, proportional to the number of subscribers, over any TCP-based file distribution program. For a large number of hosts, afpdist distributes files far more efficiently than presently available programs such as rdist and track.

5 Applications

AFDP has been implemented as two co-operating programs, which may be run on most Unix platforms and without any kernel modifications. One program, afdpjoin, handles subscriber and secretary functions, and the second, afpsend, handles publisher functions. The publisher can also specify an external program to be run by each subscriber after receiving the file. This allows AFDP to be used as a building block in the construction of larger distributed applications, such as the ones we now present.

5.1 The afpdist Program

In large workstation clusters, there are often many files that must be updated frequently. This task

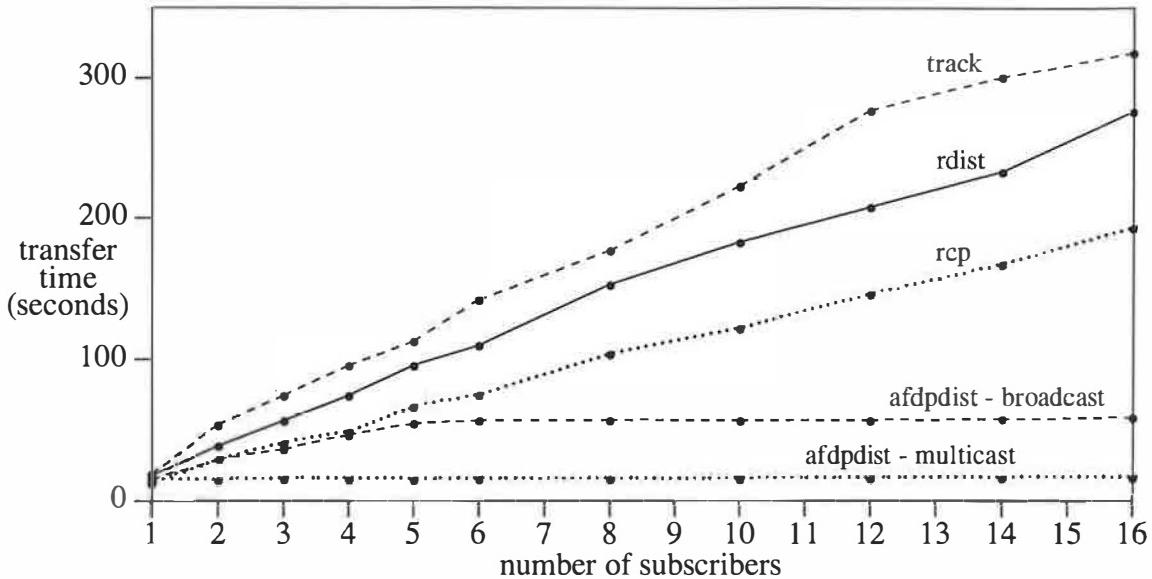


Figure 4: File distribution time vs. number of subscribers for various file update programs. In each case, the entire contents of a 7.0 Mbytes directory were transferred to each host. Peak Ethernet utilizations for the various programs were: **rcp** 95%, **rdist** 88%, **track** 74%, **afpdist** (multicast mode) 90%, **afpdist** (broadcast mode) 15%. The decreased ethernet utilization of **afpdist** in broadcast mode vs. multicast and unicast modes is due to the limiting effect of system calls on the smaller broadcast packets. Also note that the original version of **rdist** was used for these tests.

can be automated with programs such as **rdist** or **track**, but their use of point-to-point communication can be very inefficient if many of the hosts are on the same local area network. For example, on the ECF cluster, an 800 Kbyte */etc/passwd* file is tracked from the master server to 40 client systems every five minutes.

An ideal solution would be to utilize AFDP as the transport layer protocol of **track** or **rdist**, an idea we have discussed with Michael Cooper, who has been tuning the performance of the latter. Cooper agrees, but notes that this would require major changes to **rdist**. Therefore, we feel that this merge would best be accomplished as part of the re-write he is planning [3].

In the meantime, we developed our own **rdist**-like program, **afpdist**, as a proof-of-concept. This allowed us to benchmark AFDP against other file distribution applications and to demonstrate the suitability of our protocol to such tasks.

While beta testing AFDP, John DiMarco <jdd@cdf.toronto.edu> reported:

Performance was impressive. **afpsend** of */etc/passwd* to 68 machines took twenty seconds. The equivalent **rdist**⁴ took 251 seconds.

⁴John DiMarco is making use of the recent enhance-

For the following discussion, we adopt terminology from **track**. An *slist* is a subscription list, containing a list of files and directories to be distributed, and a *statsfile* contains file size and modification time information for each entry of the *slist*.

When **afpdist** is invoked on the master machine, it will generate a *statsfile* for the specified *slist* and send this file to all subscribers using the AFDP protocol. Upon receipt of the master *statsfile*, each subscriber will generate its own local *statsfile* and compare the two to produce a list of requested files. This list is then sent back to the master over a TCP connection. When the master has received replies from all subscribers, it combines their request lists and generates a bundle of all the necessary files. The bundle is then distributed by AFDP to all subscribers, which unbundle the files they requested.

This scheme works well provided that most systems require the same files, such as after a software upgrade on the master system. If only one host needs files, then this can be very wasteful. In this case, it would be better to unicast files to the one host.

The current implementation of **afpdist** is nei-

ments to **rdist**, provided by Cooper, which perform multiple TCP transfers in parallel.

ther feature-rich, nor does it include as many configuration options as `rdist` or `track`. For simplicity, the program uses `tar` to bundle and unbundle files, and shell scripts to carry out operations whose performance is not critical. For example, the command, ‘`cat request.* | sort | uniq > bundle.list`’, is used to merge together the lists of files requested by subscribers. Also, the *slists* specification is presently very restrictive: only a single directory tree can be specified, with no provision to exclude subdirectories.

5.2 Conferencing

The ease with which AFDP supports group communication makes it ideally suited to distributed “slide-show” applications, in which files are simultaneously sent to all participants.

We have implemented an external front-end to AFDP, called `magic-cat`, which takes appropriate action based on the contents (as determined by the `file` utility) of any received files to develop sample applications of this sort. For example, `magic-cat` uses `xloadimage` to view GIF and JPEG images, and `ghostview` to display Postscript files. We have successfully used this application in Computer Science and Engineering tutorials, allowing the instructor to control the demonstration, just as in a real slide-show.

6 Conclusions

The group communication capabilities of broadcast and multicast offer an appealing alternative to repeated unicasting of data to many machines. To achieve reliability and efficiency, a retransmission scheme and rate-based flow control mechanism must be added. AFDP provides these features, allowing large amounts of data to be distributed quickly to multiple hosts on a LAN. Changes in available network bandwidth as well as subscriber capabilities are dynamically accommodated.

Our initial implementation of `afdpdist` demonstrates the potential of file distribution that is dependent only on the network and the speed of the slowest machine, rather than the number of participating hosts. Even for relatively small clusters, the improved performance of `afdpdist` over `rdist` or `track` is significant. Other applications making use of AFDP may benefit similarly.

Availability

The source code for AFDP is available through the URL <http://www.ecf.toronto.edu/afdp> or via anonymous ftp from <ftp://ecf.toronto.edu:/pub/afdp/>. Questions should be addressed to `afdp@ecf.toronto.edu`. AFDP has been tested under IRIX, SunOS, Solaris, RISC/os, Ultrix, and SVR4.2. Our sources include a library of convenience functions for writing network applications, which can be re-used in other programs.

References

- [1] Jon Postel. Transmission Control Protocol. RFC 793, USC/Information Sciences Institute, September 1981.
- [2] Daniel Nachbar. When Network File Systems Aren't Enough: Automatic Software Distribution Revisited. In *Proceedings of the Summer USENIX Conference*, Atlanta, GA, 1986.
- [3] Michael A. Cooper. Overhauling Rdist for the '90s. In *Large Installation System Administrators Workshop Proceedings (LISA VI)*, pages 1–8, Long Beach, CA, 1992.
- [4] Jeffrey Mogul. Broadcasting Internet Datagrams. RFC 919, October 1984.
- [5] Steve Deering. Host Extensions for IP Multicasting. RFC 1112, August 1989.
- [6] Jon Postel. User Datagram Protocol. RFC 768, USC/Information Sciences Institute, August 1980.
- [7] Brian Whetten, Simon Kaplan, and Todd Montgomery. A High Performance Totally Ordered Multicast Protocol. Submitted for publication, 1995.
- [8] Alan Carroll. *ConversationBulider: A Collaborative Erector Set*. PhD thesis, Department of Computer Science, University of Illinois, 1993.
- [9] D. A. Agarwal, P. M. Melliar-Smith, and L. E. Moser. Totem: A Protocol for Message Ordering in a Wide-Area Network. In *Proceedings of the First ISMM International Conference on Computer Communications and Networks*, pages 1–5, San Diego, CA, June 1992.

- [10] J. Ioannidis and G. Maguire Jr. The Coherent File Distribution Protocol. RFC 1235, June 1991.
- [11] Brian Oki, Manfred Pfuegl, Alex Siegel, and Dale Skeen. The Information Bus - An Architecture for Extensible Distributed Systems. In *Fourteenth ACM Symposium on Operating Systems Principles*, pages 58–68, Asheville, NC, December 1993.
- [12] David D. Clark, Mark L. Lambert, and Lixia Zhang. NETBLT: A Bulk Data Transfer Protocol. RFC 998, March 1987.
- [13] David R. Cheriton. The V Distributed System. *Communications of the ACM*, 31(2), March 1988.
- [14] S. Armstrong, A. Freier, and K. Marzullo. Multicast Transport Protocol. RFC 1301, February 1992.
- [15] Kenneth P. Birman and Thomas Joseph. *Exploiting Replication*. Addison-Wesley/ACM Press Series, Sape J. Mullender, ed., June 1988.
- [16] Kenneth P. Birman. The Process Group Approach to Reliable Distributed Computing. *Communications of the ACM*, 36(12):37–53, December 1993.
- [17] Eva Henriksen, Gisle Aas, and Jan B. Rydning. A Transport Protocol Supporting Multicast File Transfer over Satellite Links. In *Proceedings of Eleventh IEEE Phoenix Conference on Computers and Communications (IPCCC)*, pages 590–596, Scottsdale, Arizona, April 1992.
- [18] M.L. Powell and D. L. Presotto. Publishing: A Reliable Broadcast Communication Mechanism. In *Proceedings of the 9th Symposium on Operating Systems Principles*, pages 100–109, New York, 1983.
- [19] David R. Cheriton. VMTP as the Transport Layer for High-Performance Distributed Systems. *IEEE Communications Magazine*, 27(6), June 1989.
- [20] W. Richard Stevens. *UNIX Network Programming*. Prentice Hall, 1990.
- [21] Jeremy R. Cooperstock and Steve Kotsopoulos. Exploiting Group Communications for Reliable High Volume Data Distribution. In

Proceedings of IEEE Pacific Rim Conference on Communications, Computers, Visualization, and Signal Processing, Victoria, BC, May 1995.

Biographical Information

Jeremy Cooperstock received the B.A.Sc. in Computer Engineering from the University of British Columbia, Vancouver, Canada, in 1990 and the M.Sc. in Computer Science from the University of Toronto, Toronto, Canada, in 1992. He is currently working towards the PhD in Electrical and Computer Engineering at the University of Toronto. From 1987 to 1988, he worked at IBM Research in Haifa, Israel, and in 1989, at the IBM T.J. Watson Research Center in Yorktown Heights, New York. His research interests include reactive environments, learning in robotic and autonomous systems, communication in distributed systems, and competitive analysis of trading strategies. He can be contacted via email at jer@dgp.toronto.edu.

Steve Kotsopoulos is a Master's student in the Department of Electrical and Computer Engineering at the University of Toronto. Since 1988, he has been a systems programmer at the university's Engineering Computing Facility. His interests include networking, security and distributed systems. He received his B.A.Sc. in Electrical Engineering from the University of Toronto in 1988.

In his spare time, Steve enjoys snowboarding and skateboarding. He can be contacted via email at steve@ecf.toronto.edu.

THE USENIX ASSOCIATION

USENIX is the UNIX and Advanced Computing Systems Technical and Professional Association. Since 1975, the USENIX Association has brought together the community of engineers, scientists, and technicians working on the cutting edge of the computing world. The USENIX conferences and technical symposia have become the essential meeting grounds for the presentation and discussion of the most advanced information on the developments of all aspects of computing systems.

USENIX and its members are dedicated to:

- problem-solving with a practical bias,
- fostering innovation and research that works,
- communicating rapidly the results of both research and innovation,
- providing a neutral forum for the exercise of critical thought and the airing of technical issues.

USENIX holds an annual technical conference, an annual system administration conference co-sponsored with SAGE, and single topic symposia throughout the year. It publishes *login*: a bi-monthly newsletter; *Computing Systems*, a quarterly technical journal published in association with The MIT Press; and conference proceedings for each of its conferences and symposia. It also sponsors local and special technical groups relevant to the UNIX environment as well as participating in various standards efforts.

SAGE, the System Administrators Guild

The System Administrators Guild, a Special Technical Group within the USENIX Association, is dedicated to the recognition and advancement of system administration as a profession. To join SAGE, you must be a member of USENIX. SAGE activities include the publishing of *Job Descriptions for System Administrators*, edited by Tina Darmohray; "SAGE News", a regular feature in *login*: The System Administrator Profile, an annual survey of system administrator salaries and responsibilities; support of working groups; and an archive site for papers from the System Administration Conferences.

Member Benefits:

- Free subscription to *login*: the Association's bi-monthly newsletter featuring technical articles, a worldwide calendar of events, SAGE News, media reviews, summaries of conferences, Snitch Reports from the USENIX representative and others on various ANSI, IEEE, and ISO standards efforts, and much more.
- Free subscription to *Computing Systems*, a refereed technical quarterly published with The MIT Press.
- Discounts on registration for technical sessions at all USENIX conferences and symposia.
- Discounts on proceedings from USENIX conferences and symposia.
- Access to the Online Library of all 1993-1995 published USENIX proceedings on the USENIX World Wide Web site
- Discount on the new 4.4BSD Manuals, the definitive release of the Berkeley version of UNIX with a CD-ROM published by the USENIX Association and O'Reilly & Associates, Inc.
- Savings on *The Evolution of C++: Language Design in the Marketplace of Ideas*, edited by Jim Waldo of Sun Microsystems Laboratories, the USENIX Association book published by The MIT Press.
- Special subscription rates to *UniForum Monthly*, *UniNews* and the annual *UniForum Open Systems Products Directory* and *Linux Journal*
- Savings on selected titles from McGraw-Hill, The MIT Press, Prentice Hall, John Wiley & Sons, O'Reilly and Associates, and UniForum.

Supporting Members of the USENIX Association:

Apunix Computer Services	Shiva Corporation
ANDATACO	Sun Microsystems, Inc., Sunsoft Network Products
Frame Technology Corporation	Tandem Computers, Inc.
GraphOn Corporation	UUNET Technologies, Inc
Matsushita Electrical Industrial Co., Ltd.	

SAGE Supporting Members: Enterprise Systems Management Corporation and Great Circle Associates

For further information about membership, conferences or publications, contact:

The USENIX Association
2560 Ninth Street, Suite 215
Berkeley, CA 94710 USA
Email: office@usenix.org
Phone: +1-510-528-8649
Fax: +1-510-548-5738
WWW URL: <http://www.usenix.org>

ISBN 1-880446-76-6